Breadth-first search

Breadth-first search

A breadth-first search (BFS) in a graph visits the nodes in the following order:

- First it visits some node (the *start node*)
- Then all the start node's immediate neighbours
- Then *their* neighbours
- and so on

So it visits the nodes in order of how far away they are from the start node

Implementing breadth-first search

We maintain a *queue* of nodes that we are going to visit soon

- Initially, the queue contains the start node

We also remember which nodes we've already added to the queue

Then repeat the following process:

- Remove a node from the queue
- Visit it
- Find all adjacent nodes and add them to the queue, *unless* they've previously been added to the queue

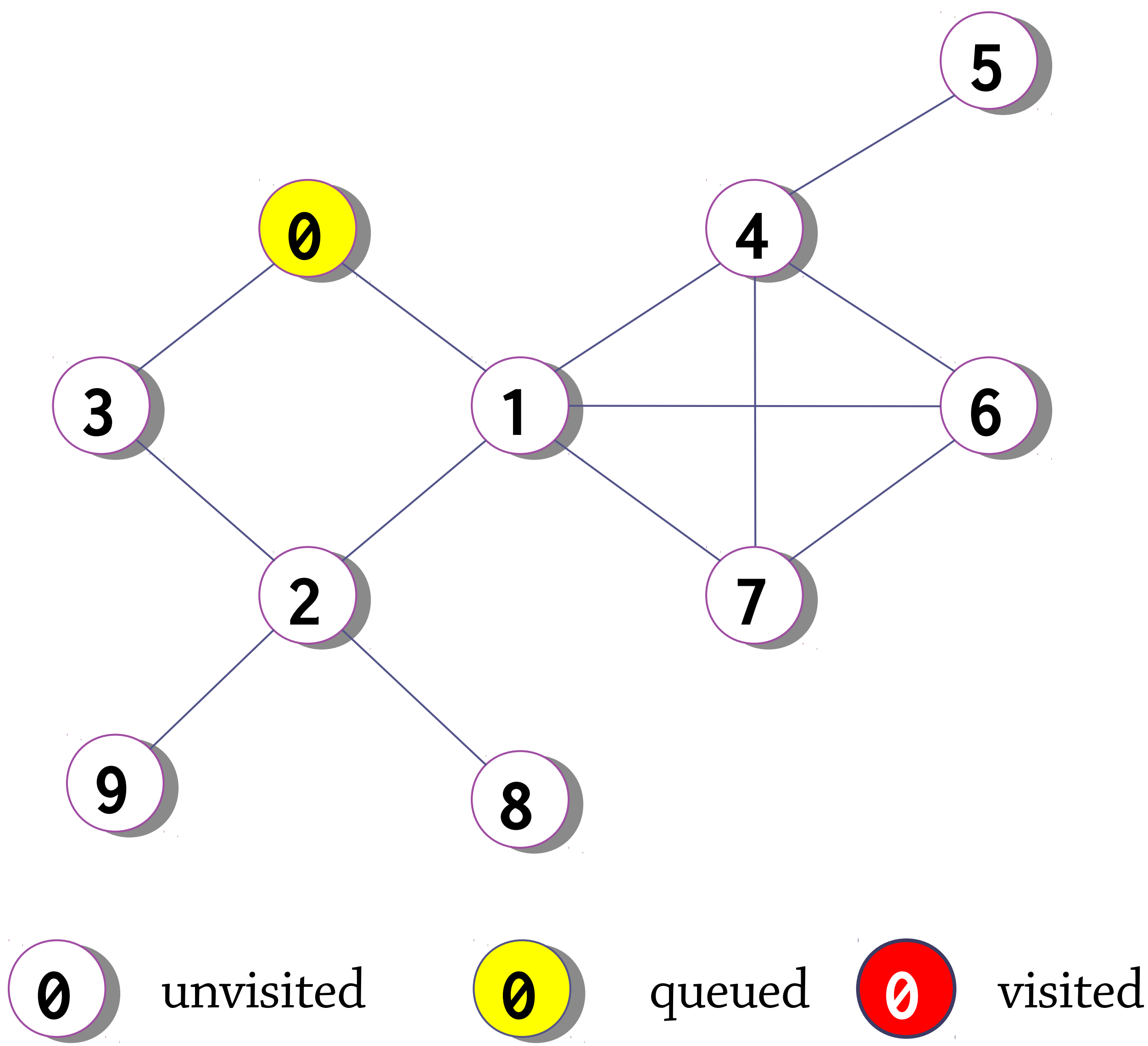
Example of a breadth-first search

Queue:

0

Visit order:

Initially,
queue contains
start node

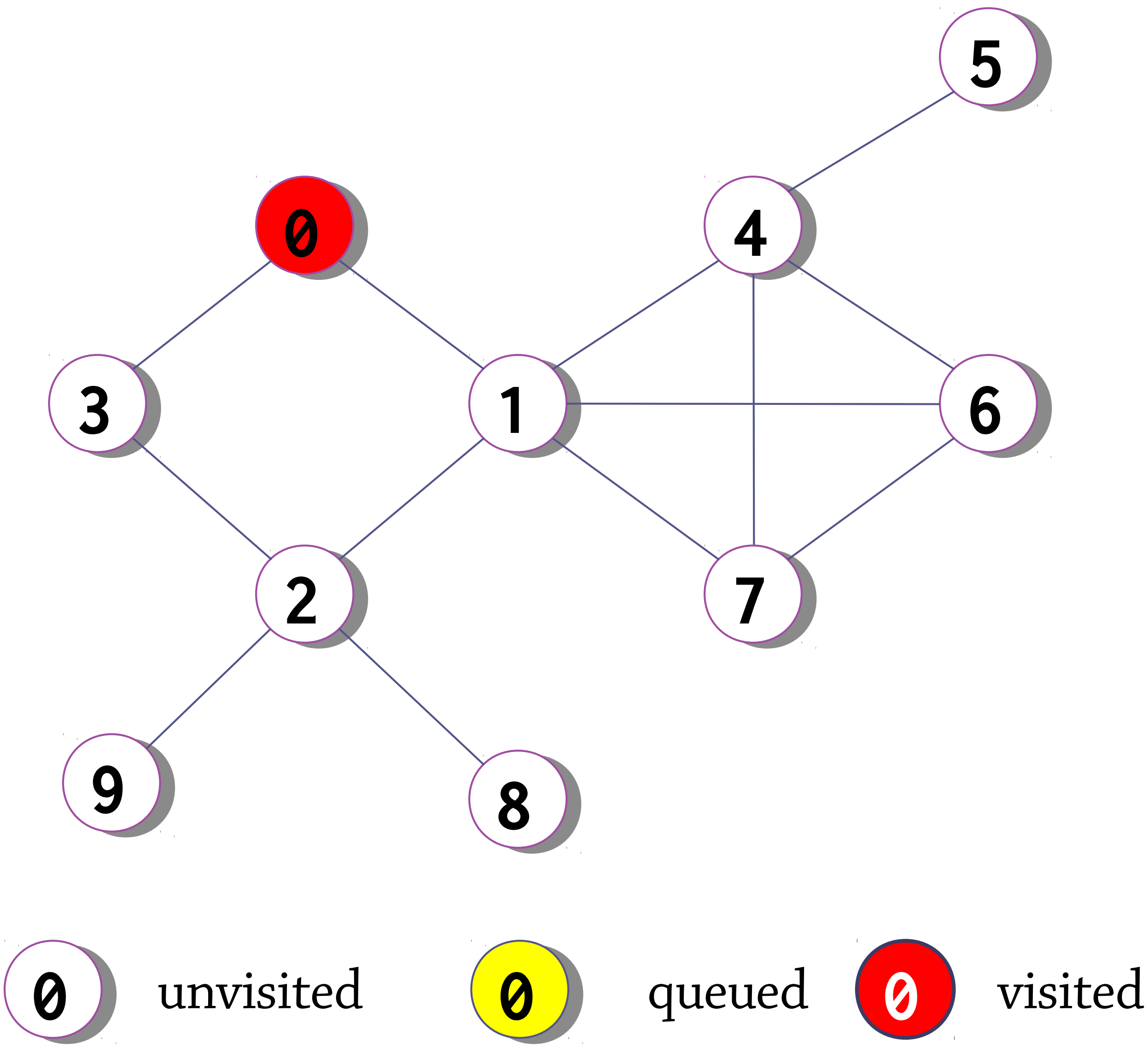


Example of a breadth-first search

Queue:

Visit order:

0



Step 1:
remove node
from queue
and visit it

Example of a breadth-first search

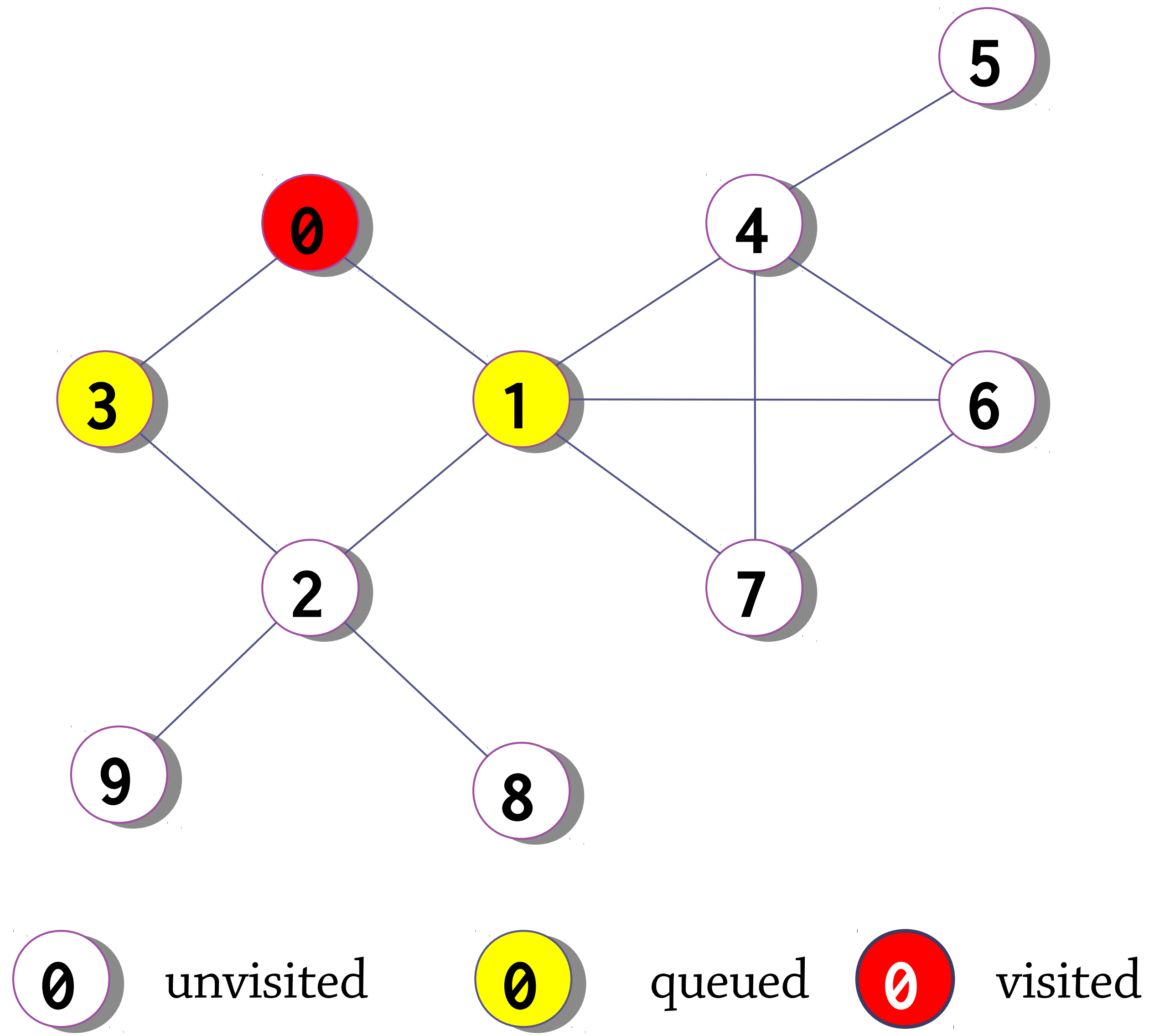
Queue:

3 1

Visit order:

0

Step 2:
add adjacent nodes to queue
(only unvisited ones)



Example of a breadth-first search

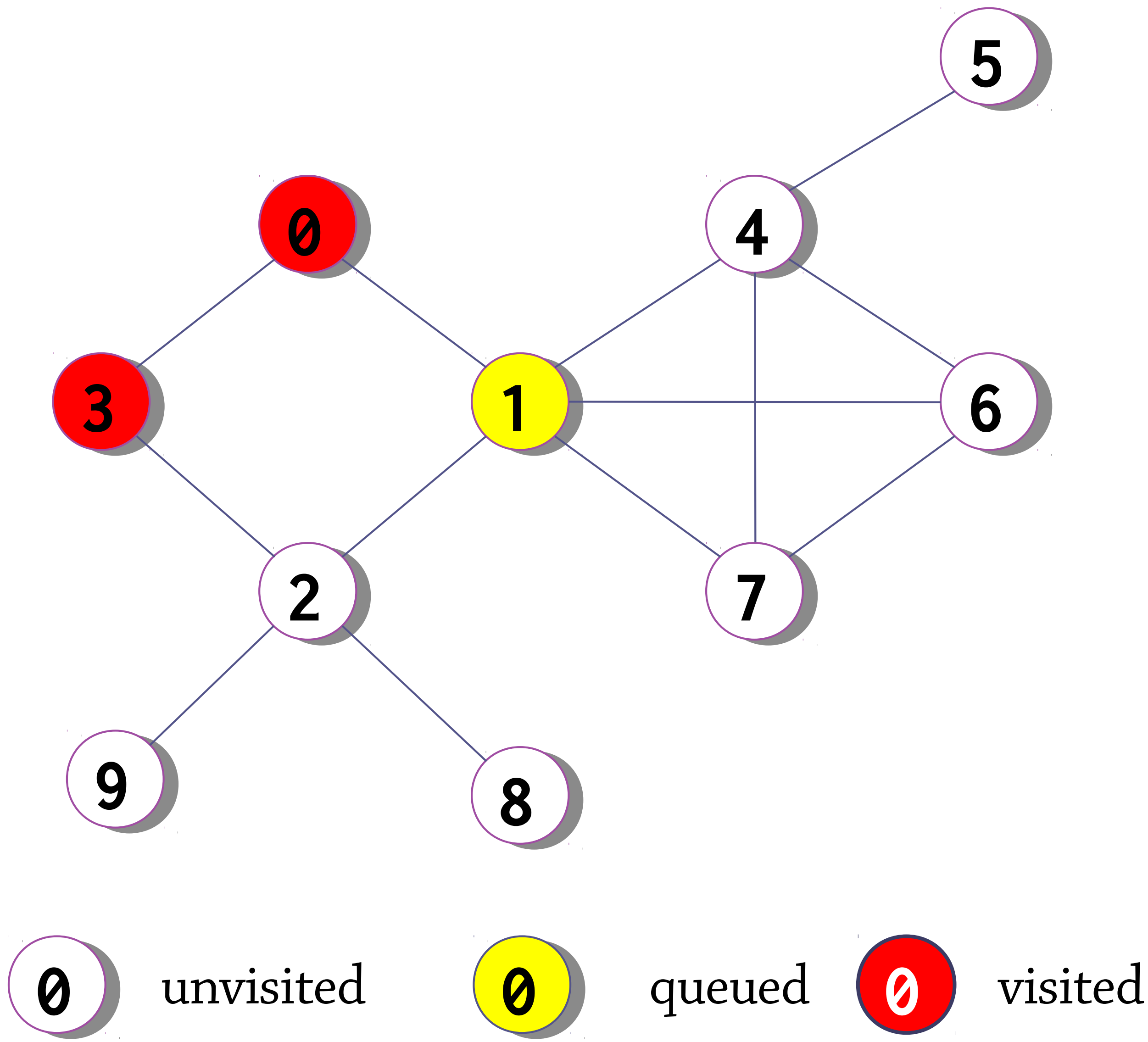
Queue:

1

Visit order:

0 3

Step 1:
remove node
from queue
and visit it



Example of a bre… arch

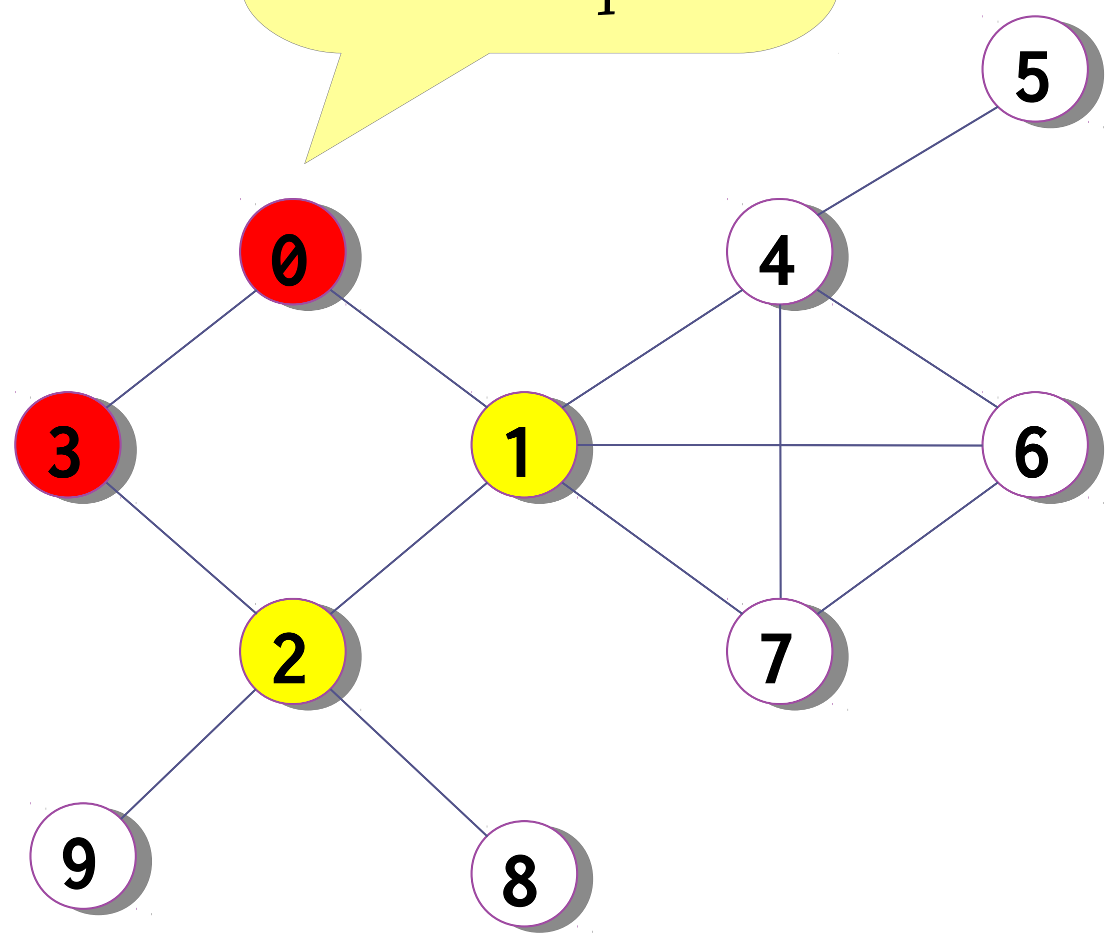
0 is already visited, so we don't add it to the queue

Queue:

1 2

Visit order:

0 3



Step 2:
add adjacent nodes to queue
(only unvisited ones)

0 unvisited 0 queued 0 visited

Example of a breadth-first search

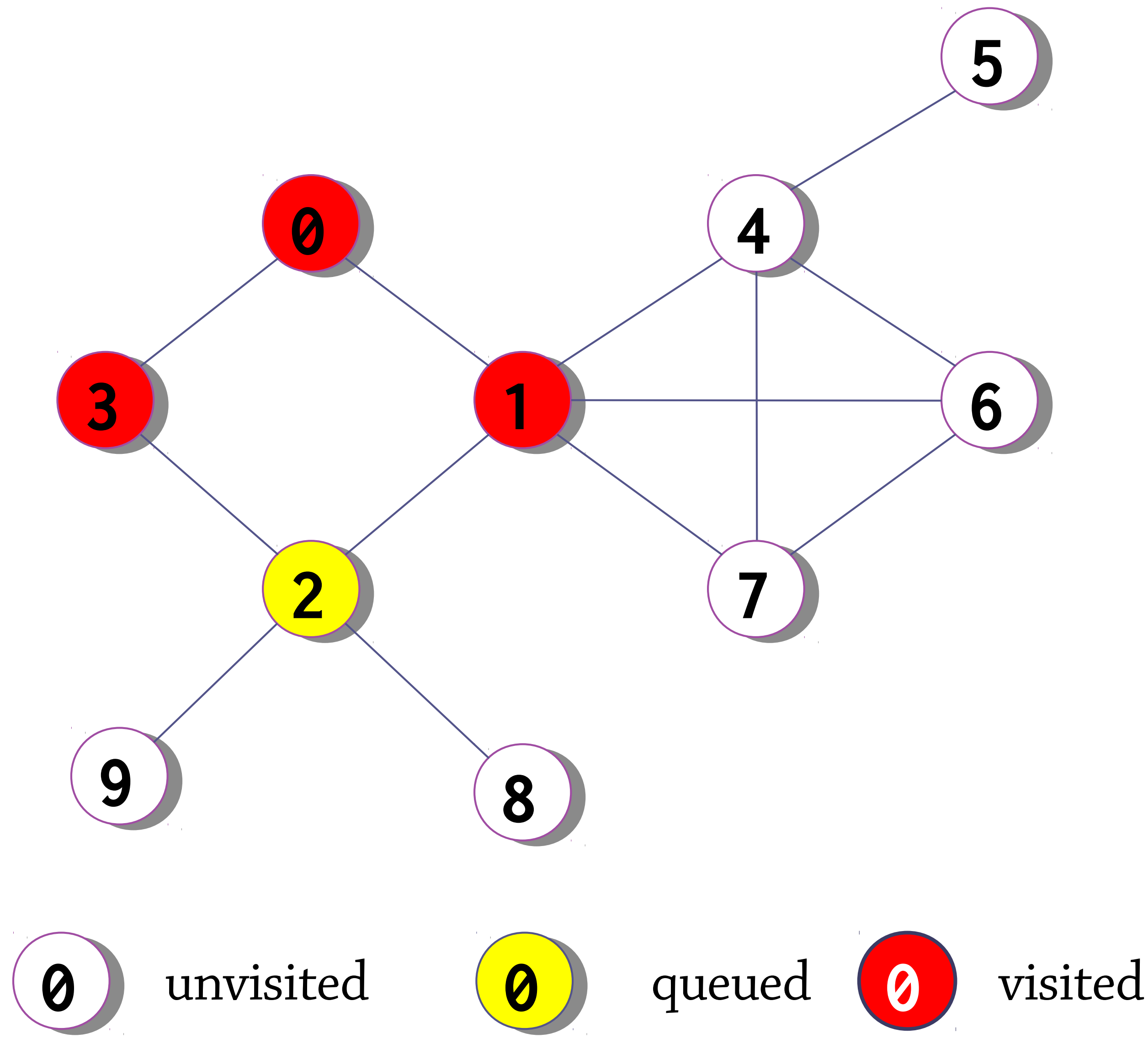
Queue:

2

Visit order:

0 3 1

Step 1:
remove node
from queue
and visit it



Example of a bre… arch

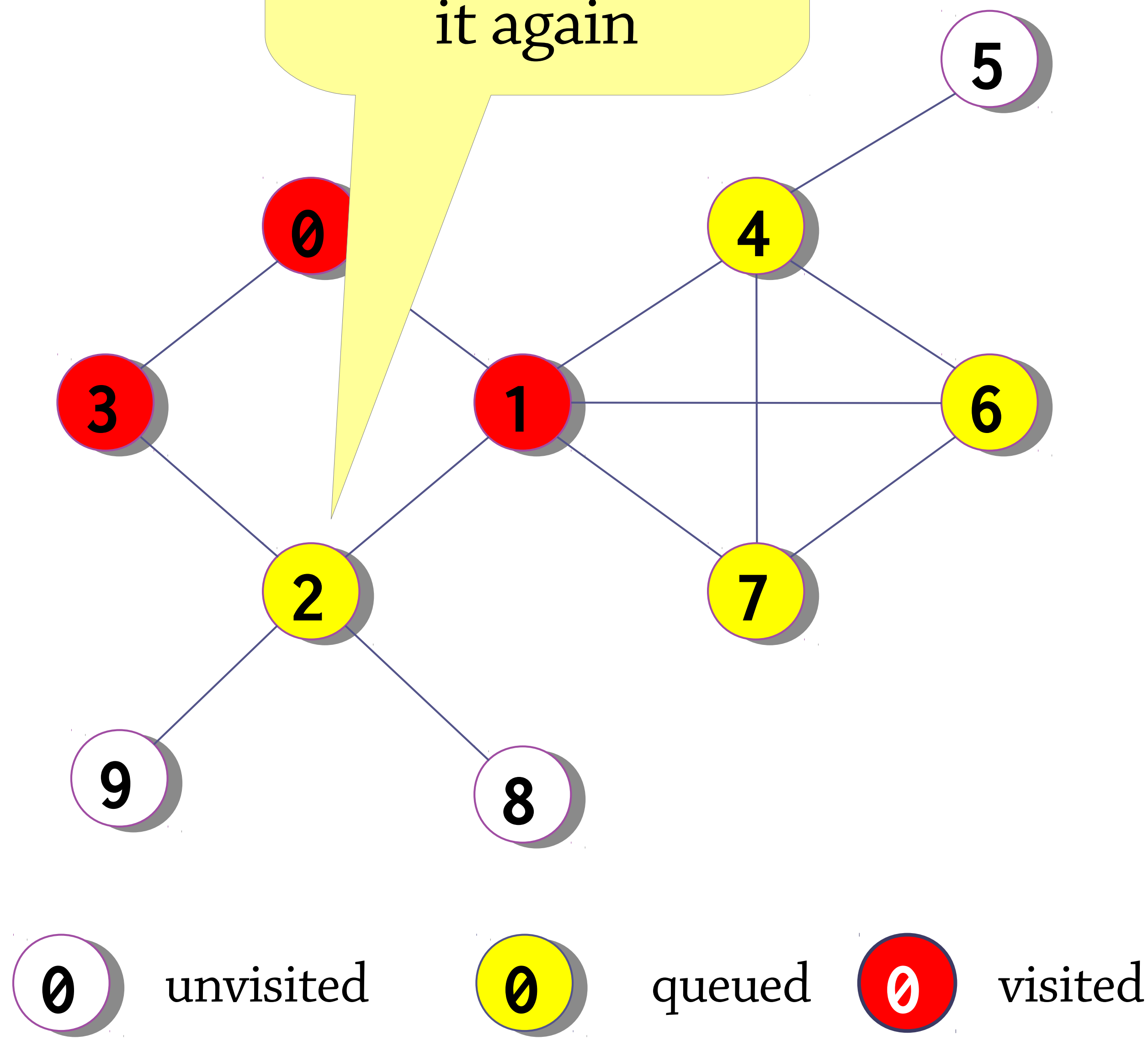
2 is already
in the queue, so
we don't add
it again

Queue:

2 4 6 7

Visit order:

0 3 1



Step 2:
add adjacent nodes
to queue
(only unvisited ones)

Example of a breadth-first search

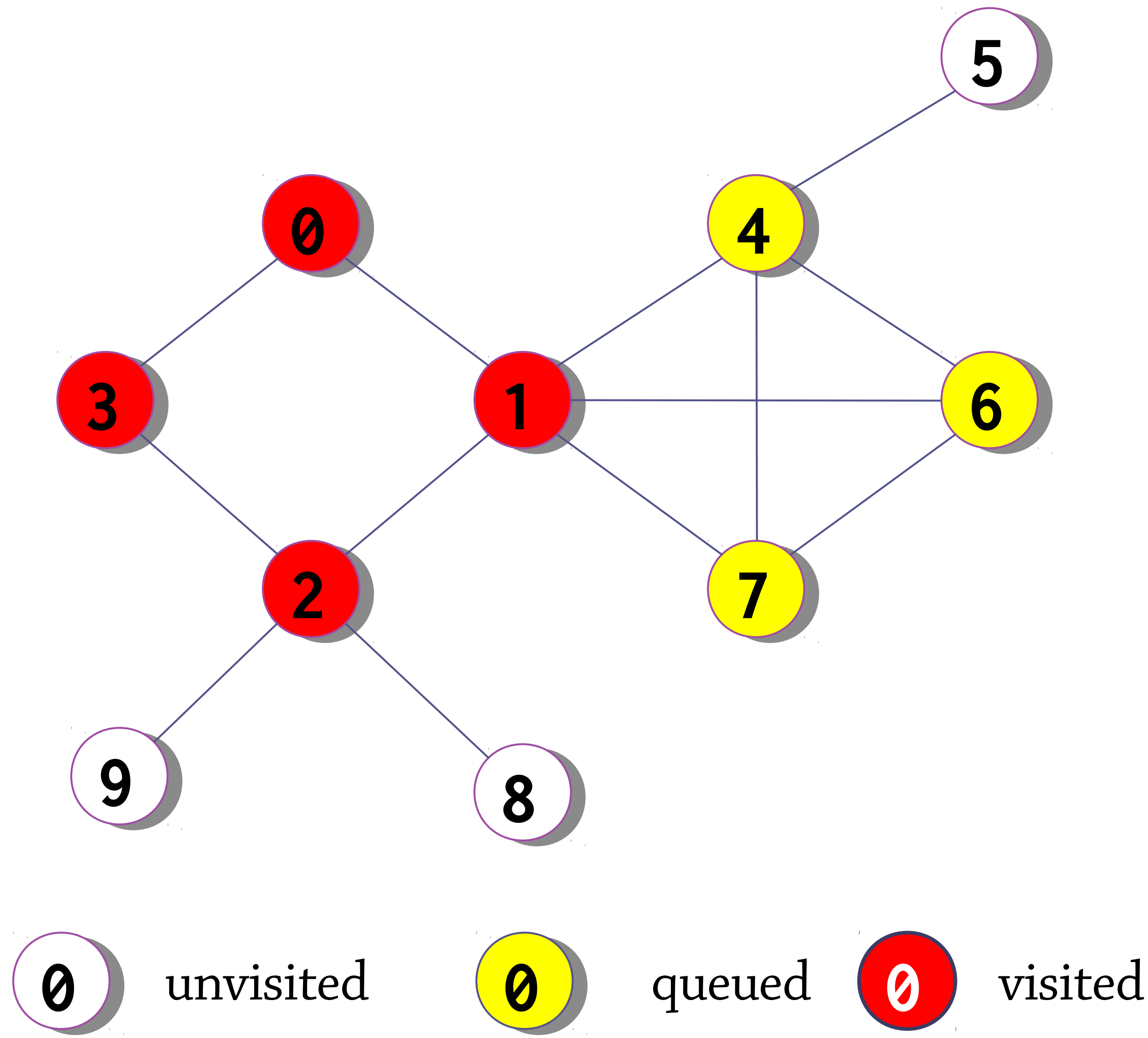
Queue:

4 6 7

Visit order:

0 3 1 2

Step 1:
remove node
from queue
and visit it



Example of a breadth-first search

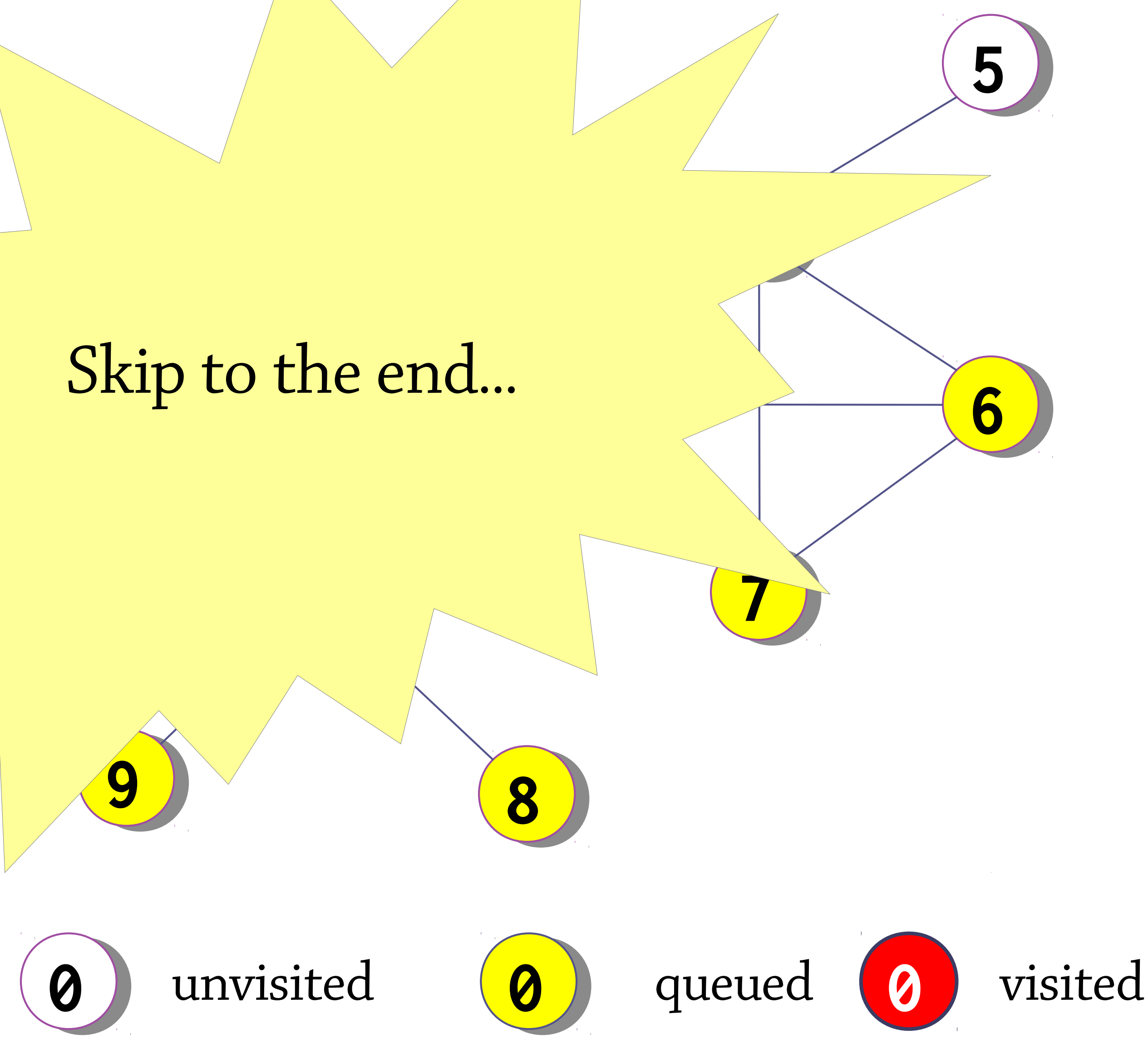
Queue:

4 6 7 9 8

Visit order:

0 3 1 2

Skip to the end...



Step 2:
add adjacent nodes
to queue
(only unvisited ones)

Example of a breadth-first search

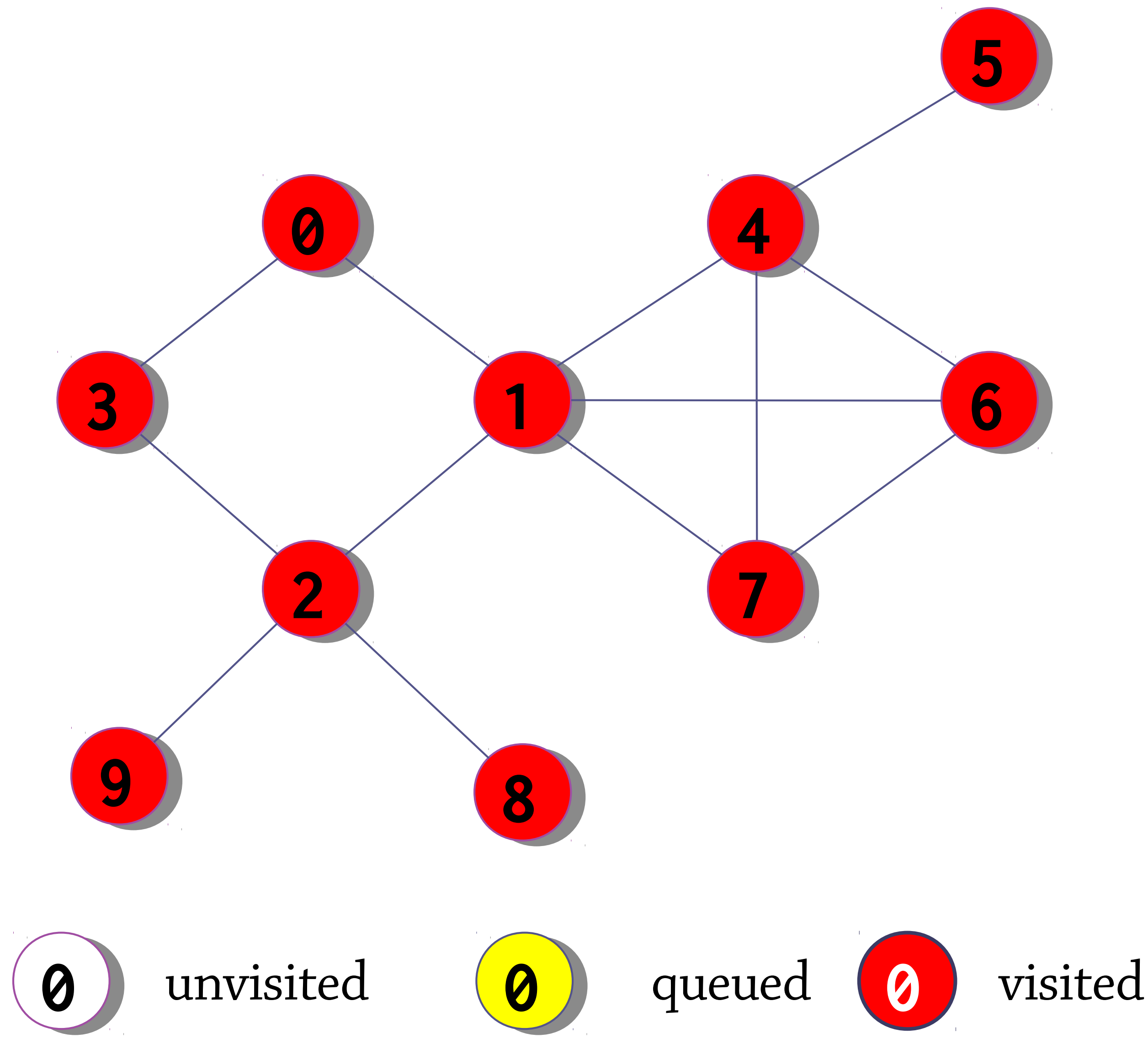
Queue:

Visit order:

0 3 1 2 4

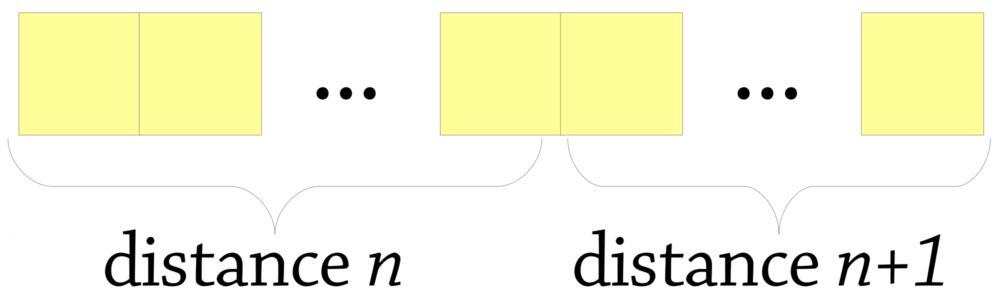
6 7 9 8 5

We reach step 1, but the queue is empty, and **we're finished!**



Why does using a queue work?

The queue in BFS always contains nodes that are *n* distance from the start node, followed by nodes that are *n+1* distance away:



When we remove the node from the head of the queue (distance *n*), we add its neighbours (distance *n+1*) to the end – so this situation remains true

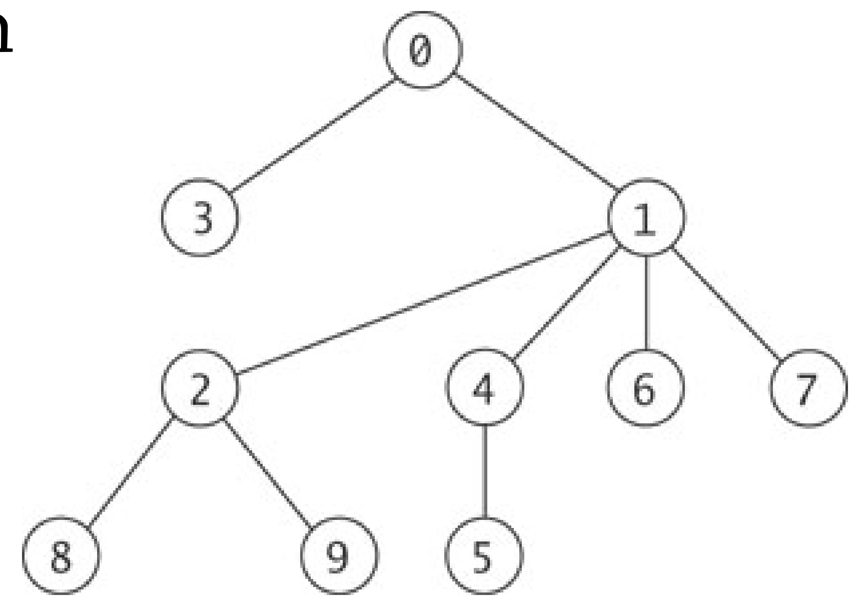
This means that we explore all nodes of distance *n* before getting to distance *n+1*

- Once we remove the first distance *n+1* node, the queue will contain nodes of distance *n+1* and *n+2*, so we go up in order of distance

Breadth-first search trees

While doing the BFS, we can record *which node we came from* when visiting each node in the graph

(we do this when adding a node to the queue)

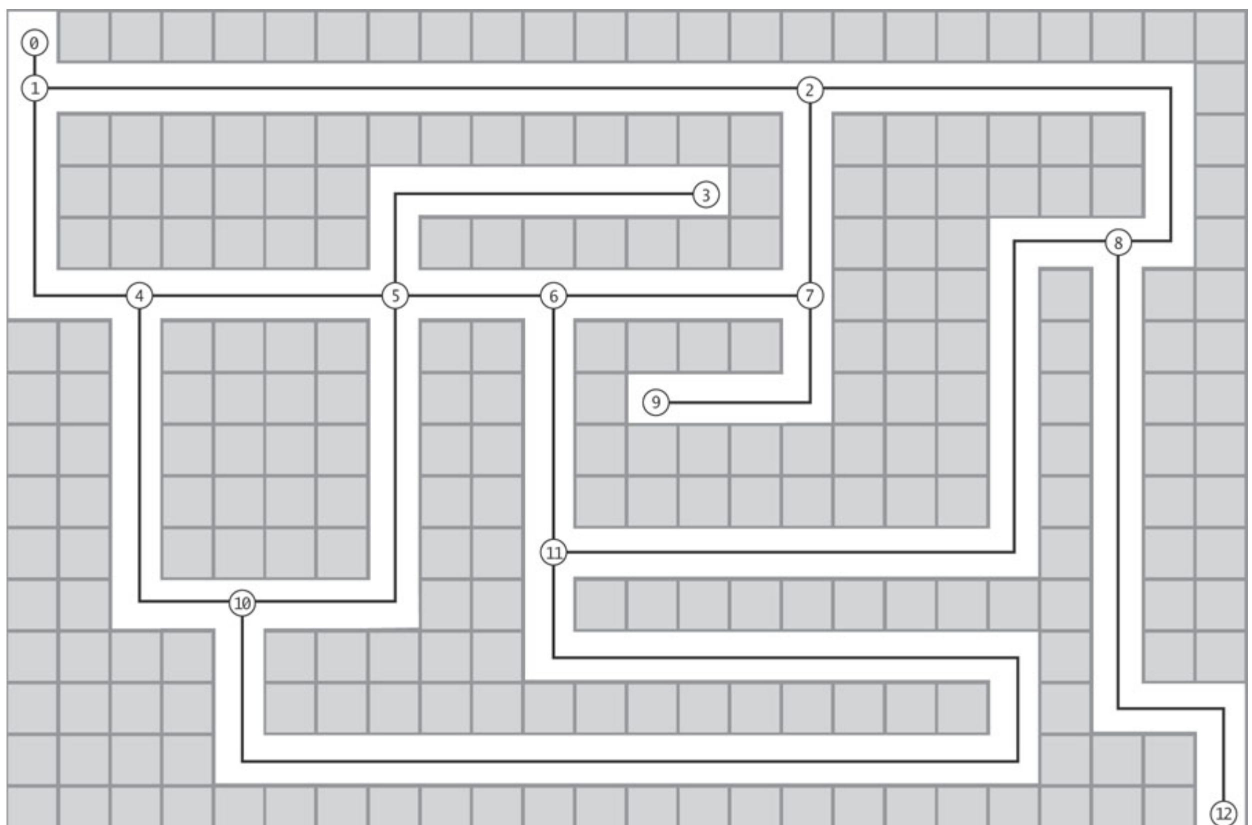


We can use this information to find the *shortest path* from the start node to any other node

We can even build the *breadth-first search tree,* which shows how the graph was explored and tells you the shortest path to all nodes

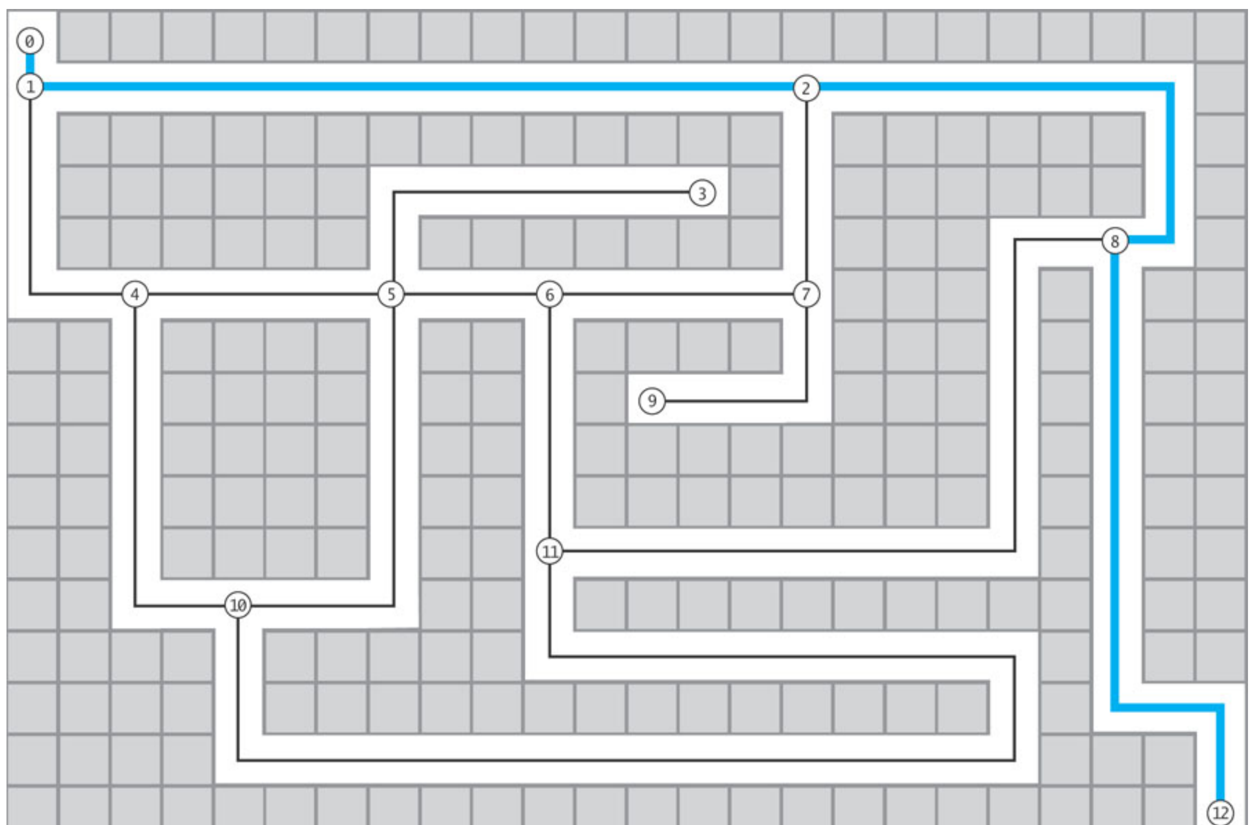
Application: unweighted shortest path

We can represent a maze as a graph – nodes are junctions, edges are paths. We want to find the simplest way (fewest choices) to get from entrance to exit



Application: unweighted shortest path

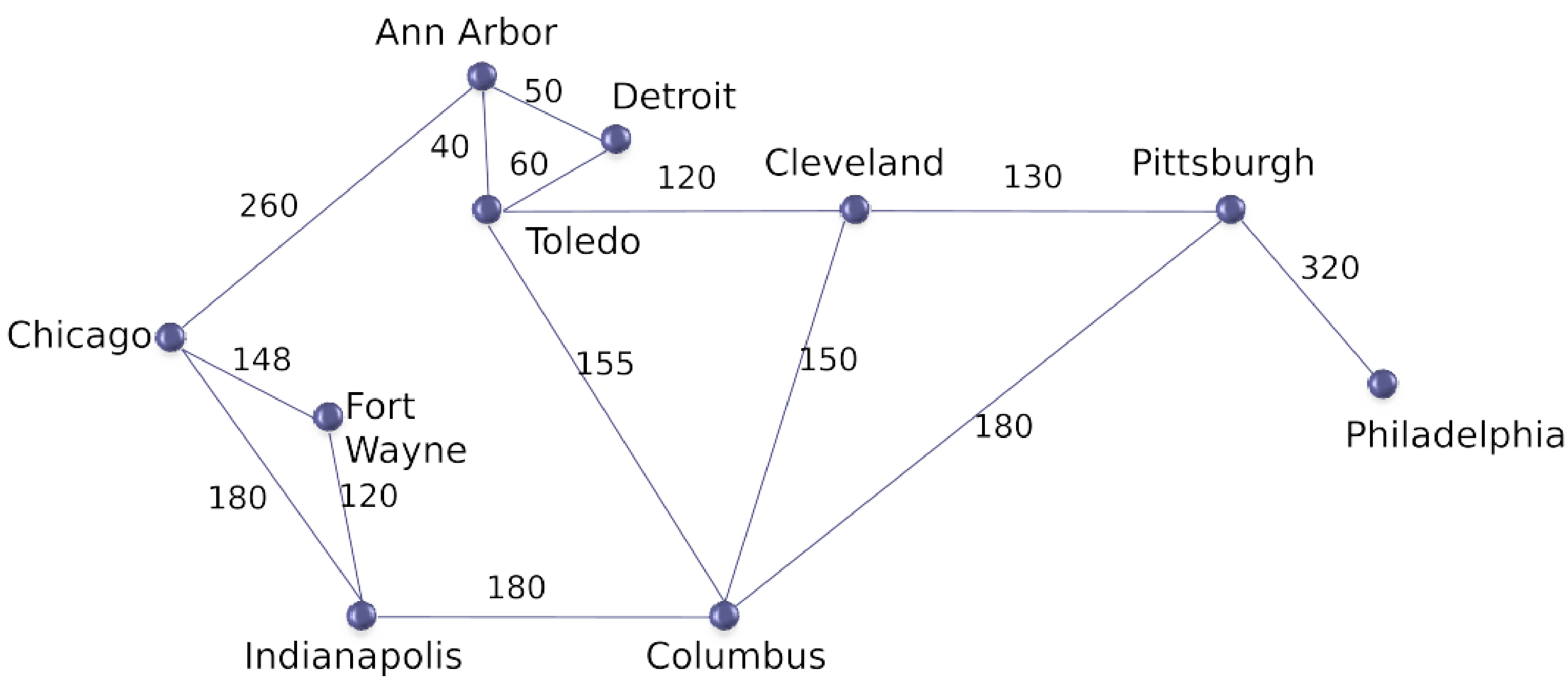
By doing a breadth-first search, and remembering how we got to each node, we will find the simplest way out of the maze



Dijkstra's algorithm
Prim's algorithm

Weighted graphs

In a *weighted graph*, each edge is labelled with a *weight*, a number:



The weight typically represents the "cost" of following the edge

The (weighted) shortest path problem

Find the *path with least total weight* from point A to point B in a weighted graph

(If there are no weights: can be solved with BFS)

Useful in e.g., route planning, network routing

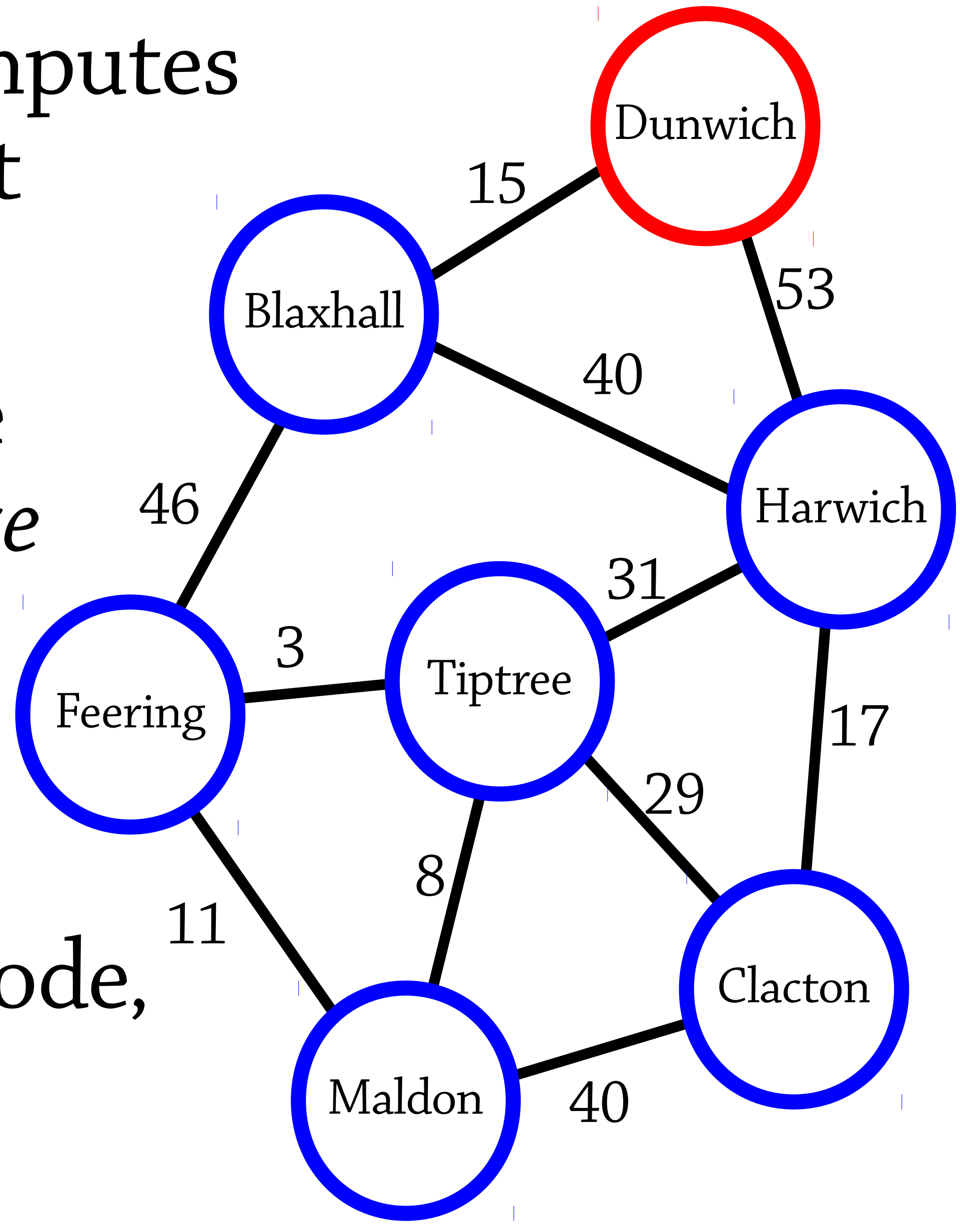
Most common approach: *Dijkstra's algorithm*, which works when all edges have positive weight

Dijkstra's algorithm

Dijkstra's algorithm computes the distance from a start node to *all other nodes*

It visits the nodes of the graph in order of *distance from the start node*, and remembers their distance

We first visit the start node, which has distance 0

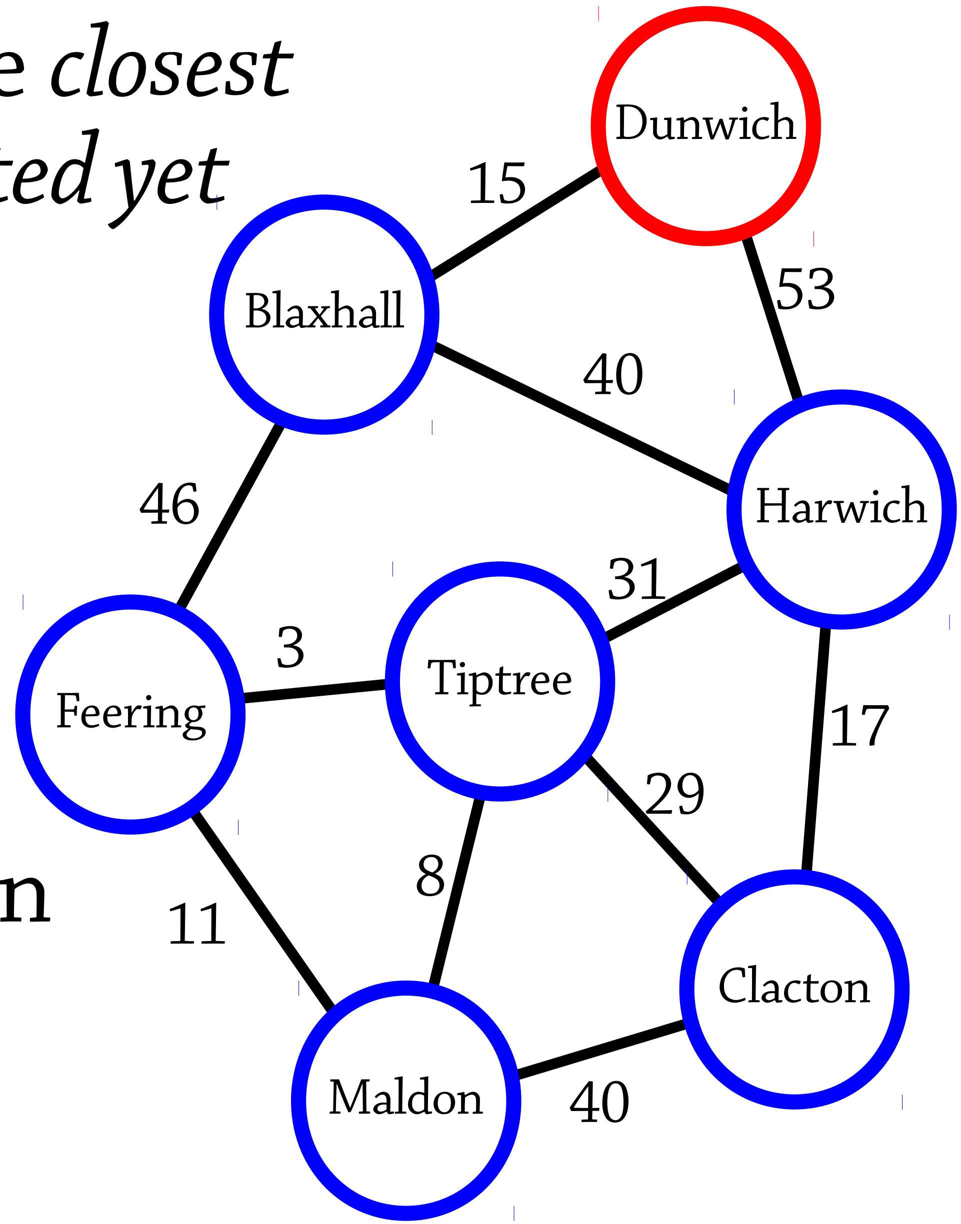


Dijkstra's algorithm

At each step we visit the *closest node that we haven't visited yet*

This node must be adjacent to a node we *have* visited (why?)

By looking at the outgoing edges from the visited nodes, we can find the closest unvisited node



Dijkstra's algorithm

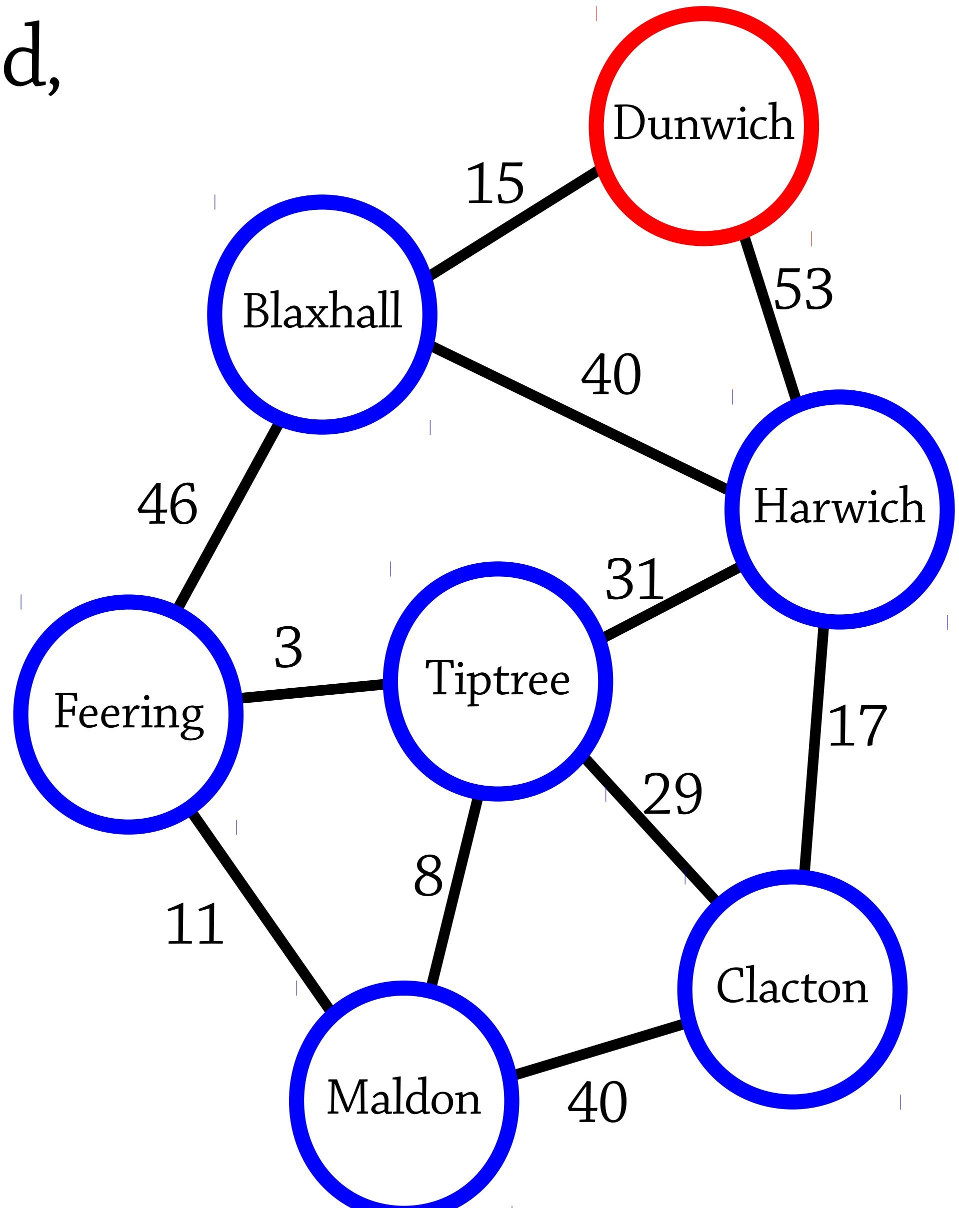
For each node *x* we've visited, and each edge x → y, where *y* is unvisited:

- Add the distance to *x* and the weight of the edge *x* → *y*

Whichever node *y* has the shortest total distance, visit it!

- This is the closest unvisited node

Repeat until there are no edges to unvisited nodes



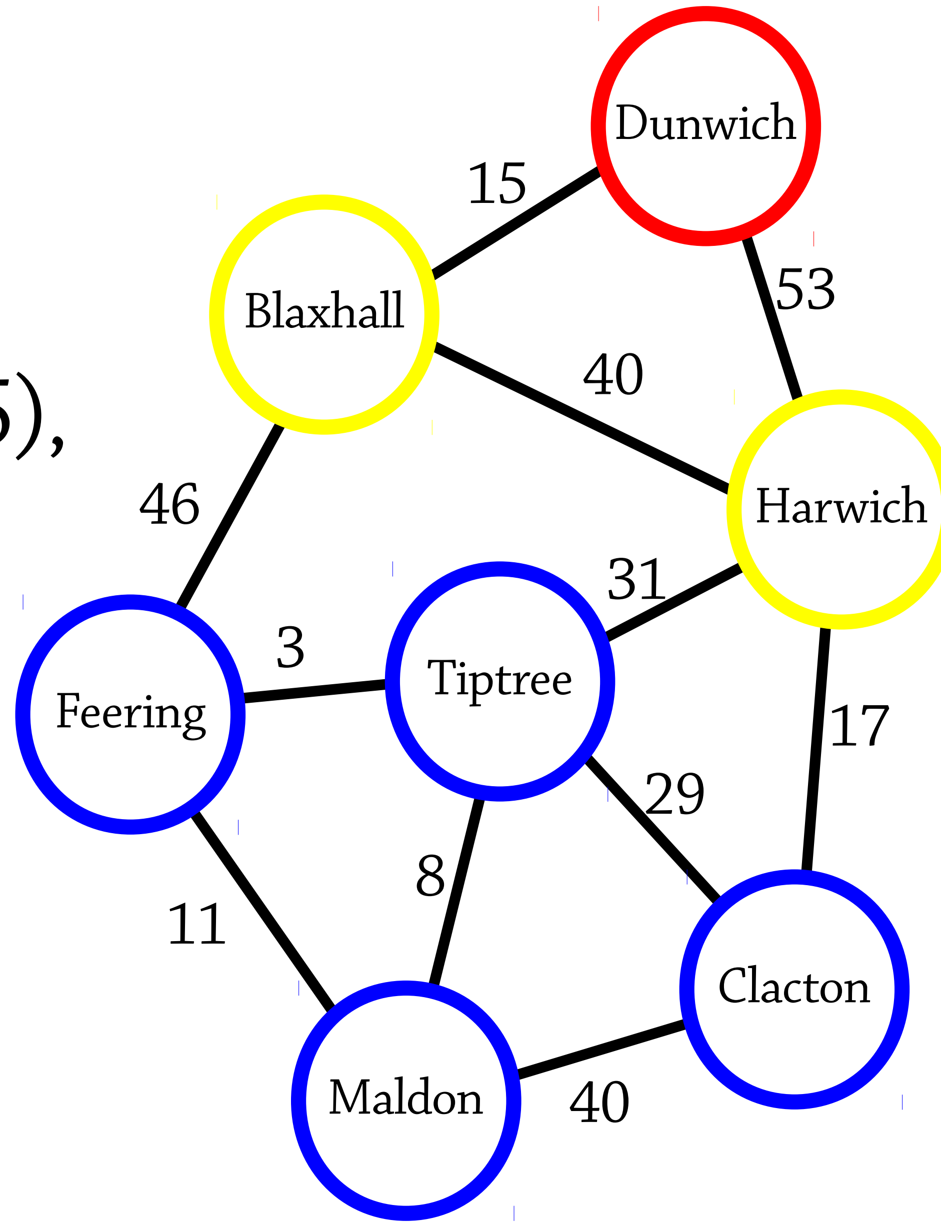
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0

Neighbours of Dunwich are Blaxhall (distance 15), Harwich (distance 53)

So visit Blaxhall (distance 15)

(Red = visited node, yellow = neighbour of visited node)



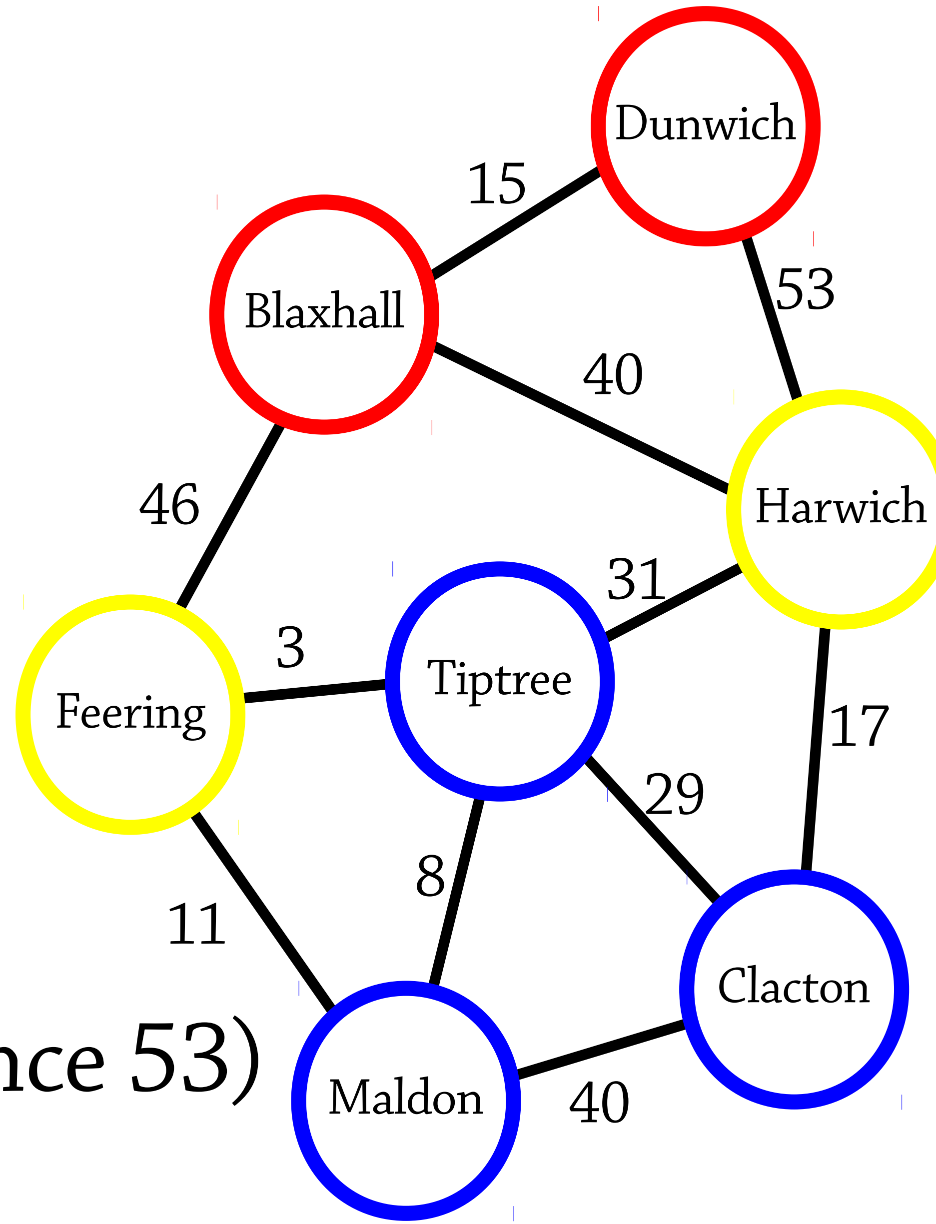
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15

Neighbours are:

- Feering (distance
  15 + 46 = 61)
- Harwich (distance 53 –
  also via Blaxhall
  15 + 40 = 55)

So visit Harwich (distance 53)



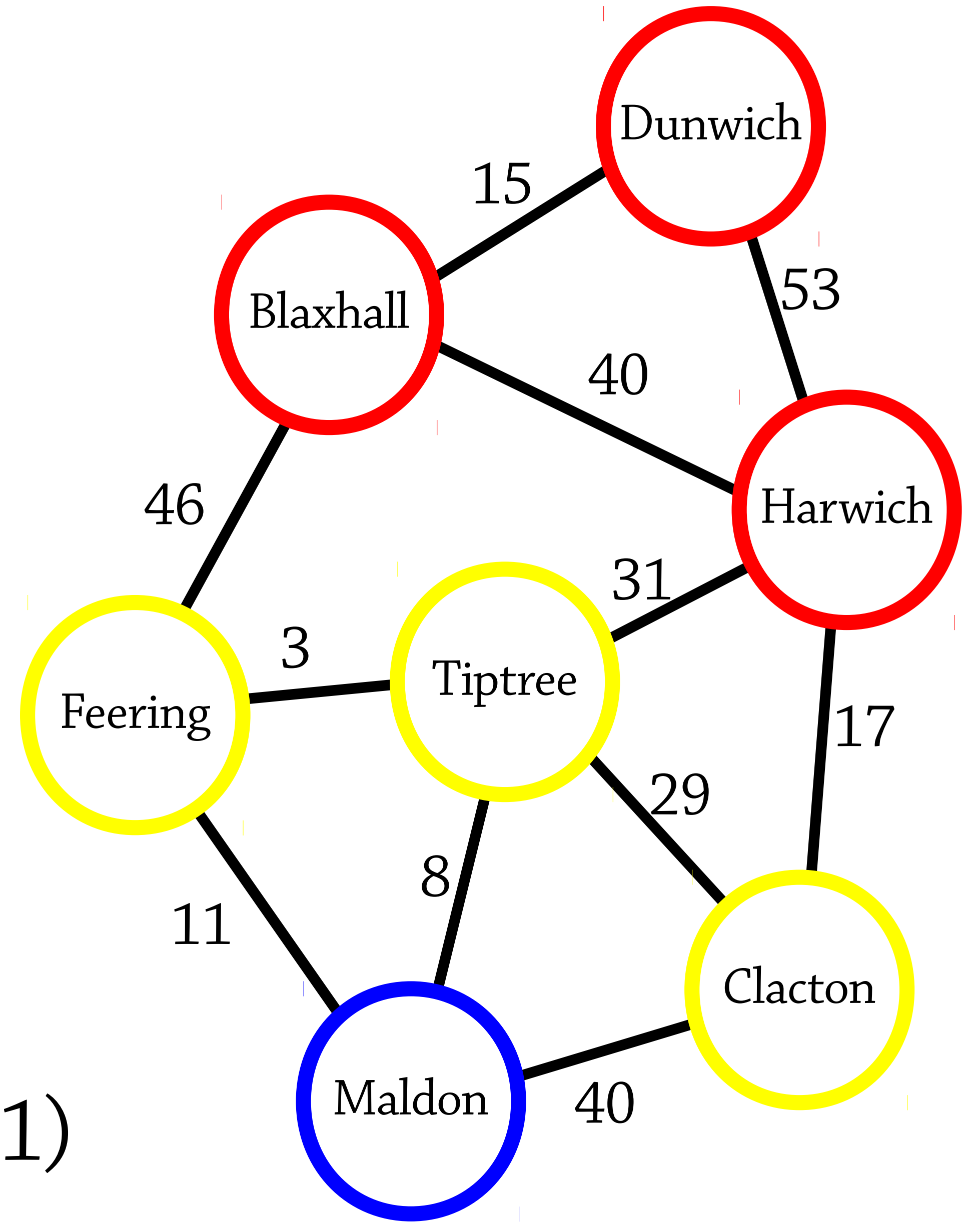
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15
Harwich distance 53

Neighbours are:

- Feering (distance 15 + 46 = 61)
- Tiptree (distance 53 + 31 = 84)
- Clacton (distance 53 + 17 = 70)

So visit Feering (distance 61)



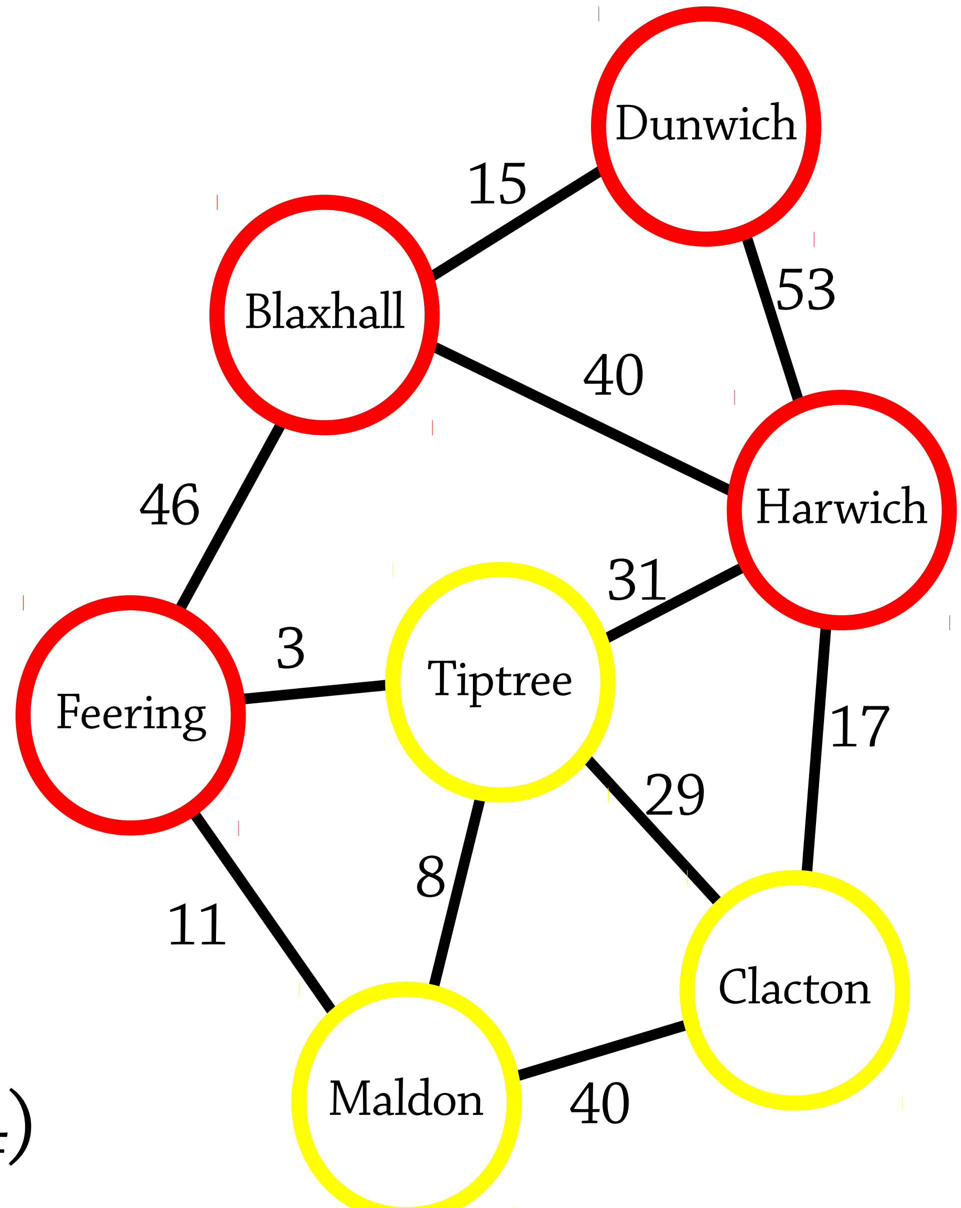
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15
Harwich distance 53
Feering distance 61

Neighbours are:

- Tiptree (distance
  61 + 3 = 64,
  also via Harwich 55 + 29 = 84)
- Clacton (distance 53 + 17 = 70)
- Malden (distance 61 + 11 = 72)

So visit Tiptree (distance 64)



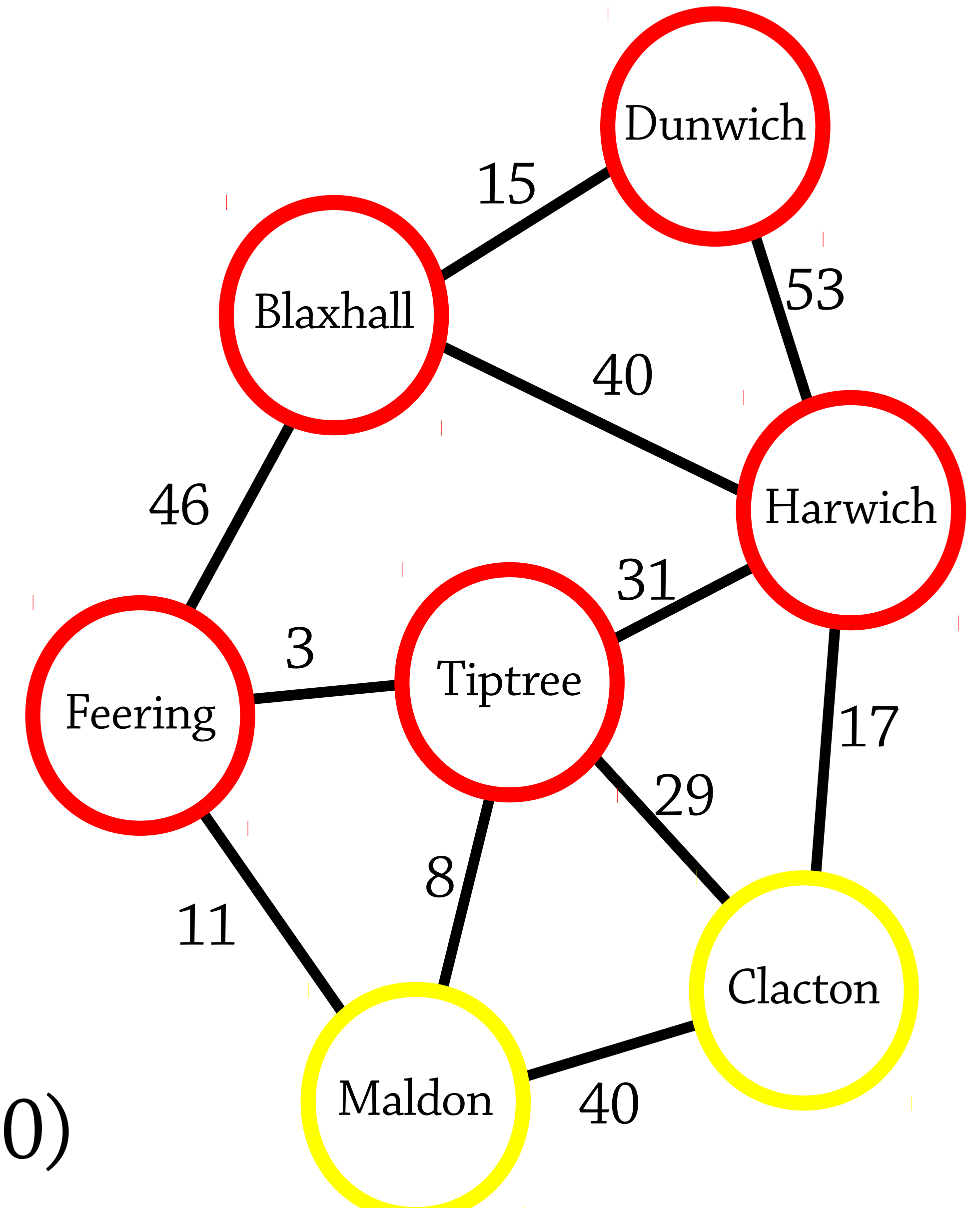
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15
Harwich distance 53
Feering distance 61
Tiptree distance 64

Neighbours are:

- Clacton (distance 53 + 17 = 70, also via Tiptree 64 + 29 = 93)
- Maldon (distance 61 + 11 = 72, also via Tiptree 64 + 8 = 72)

So visit Clacton (distance 70)



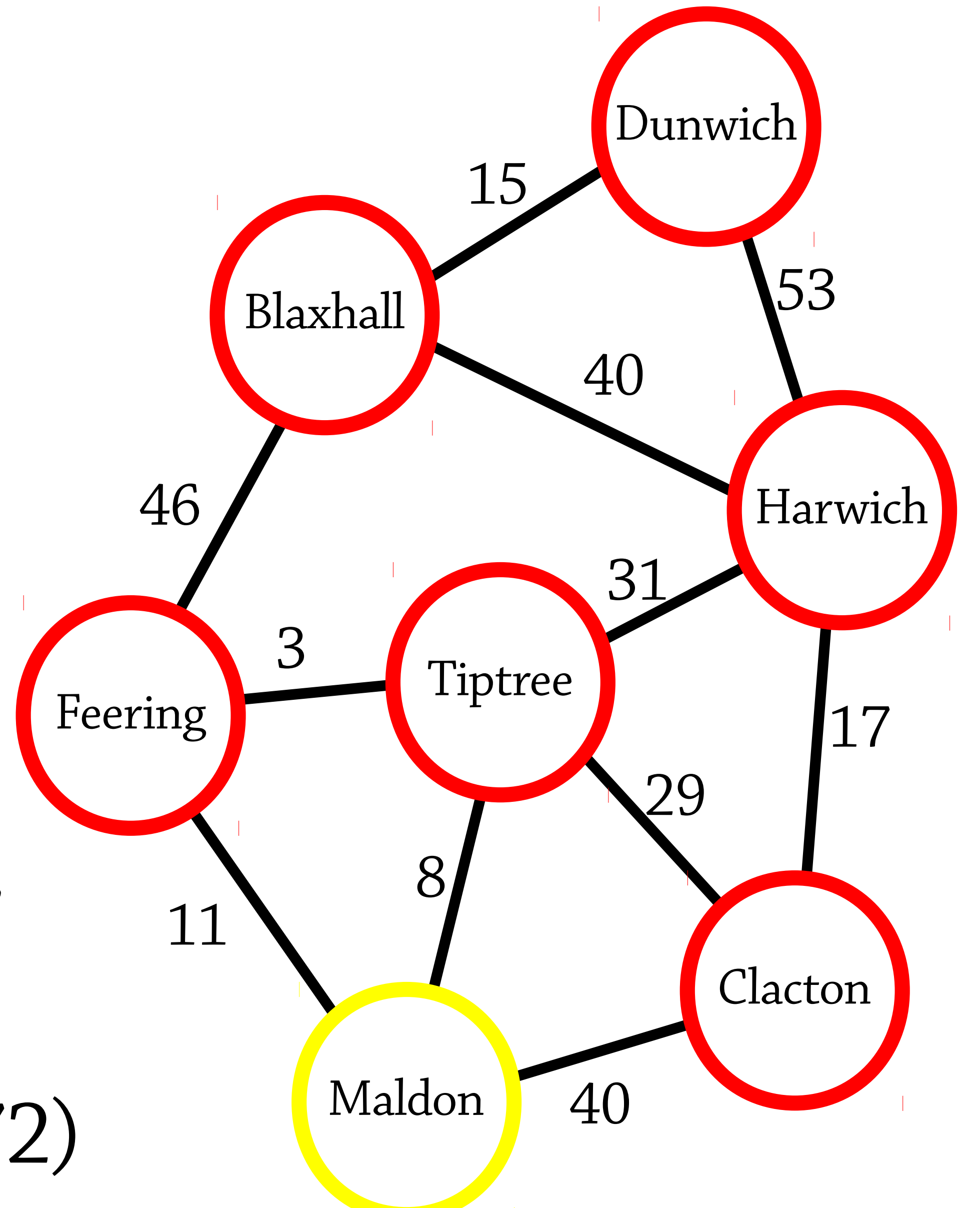
Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15
Harwich distance 53
Feering distance 61
Tiptree distance 64
Clacton distance 70

Neighbours are:

- Maldon (distance 61 + 11 = 72, also via Tiptree 64 + 8 = 72, also via Clacton 70 + 40 = 110)

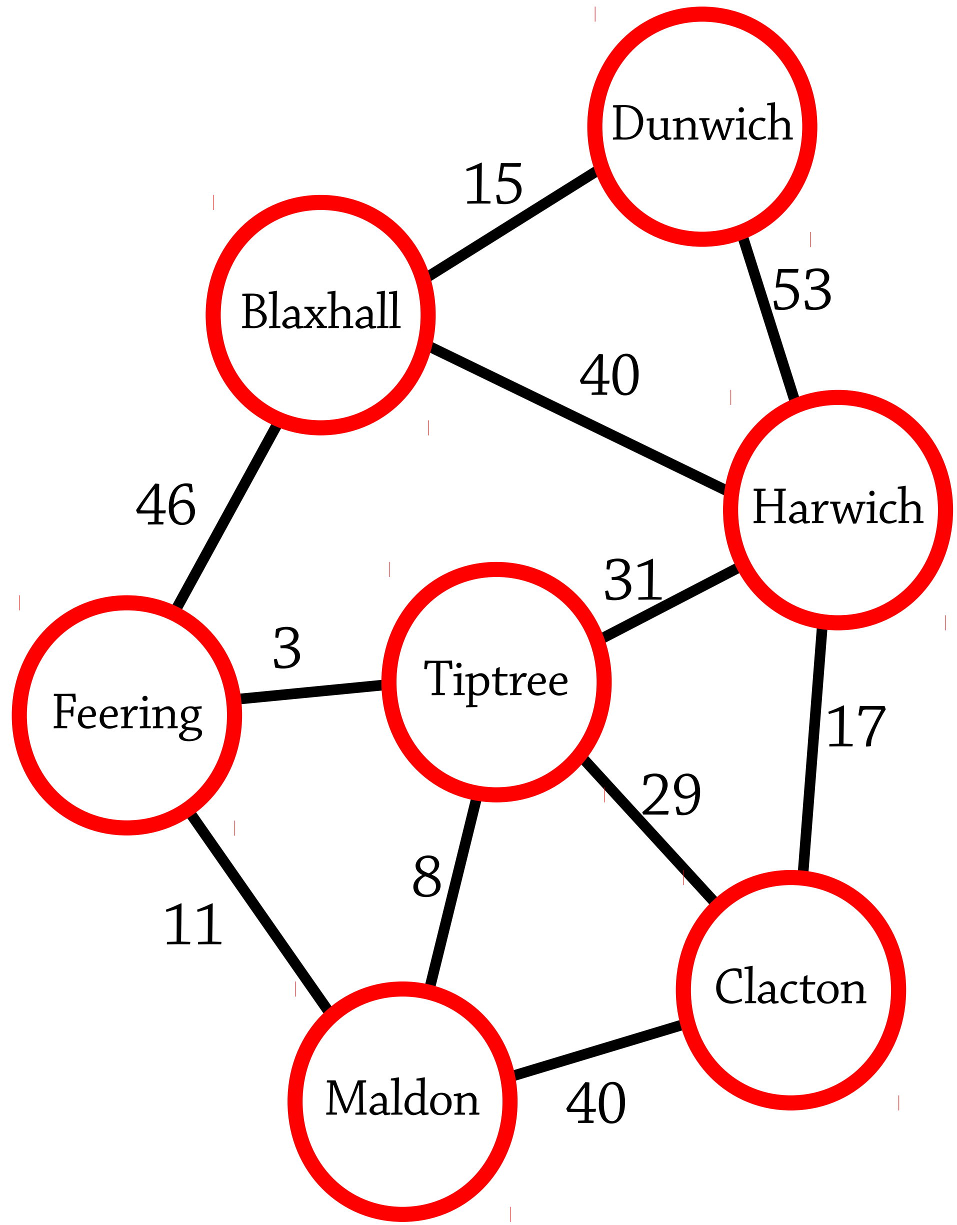
So visit Maldon (distance 72)



Dijkstra's algorithm

Visited nodes:
Dunwich distance 0
Blaxhall distance 15
Harwich distance 53
Feering distance 61
Tiptree distance 64
Clacton distance 70
Maldon distance 72

Finished!



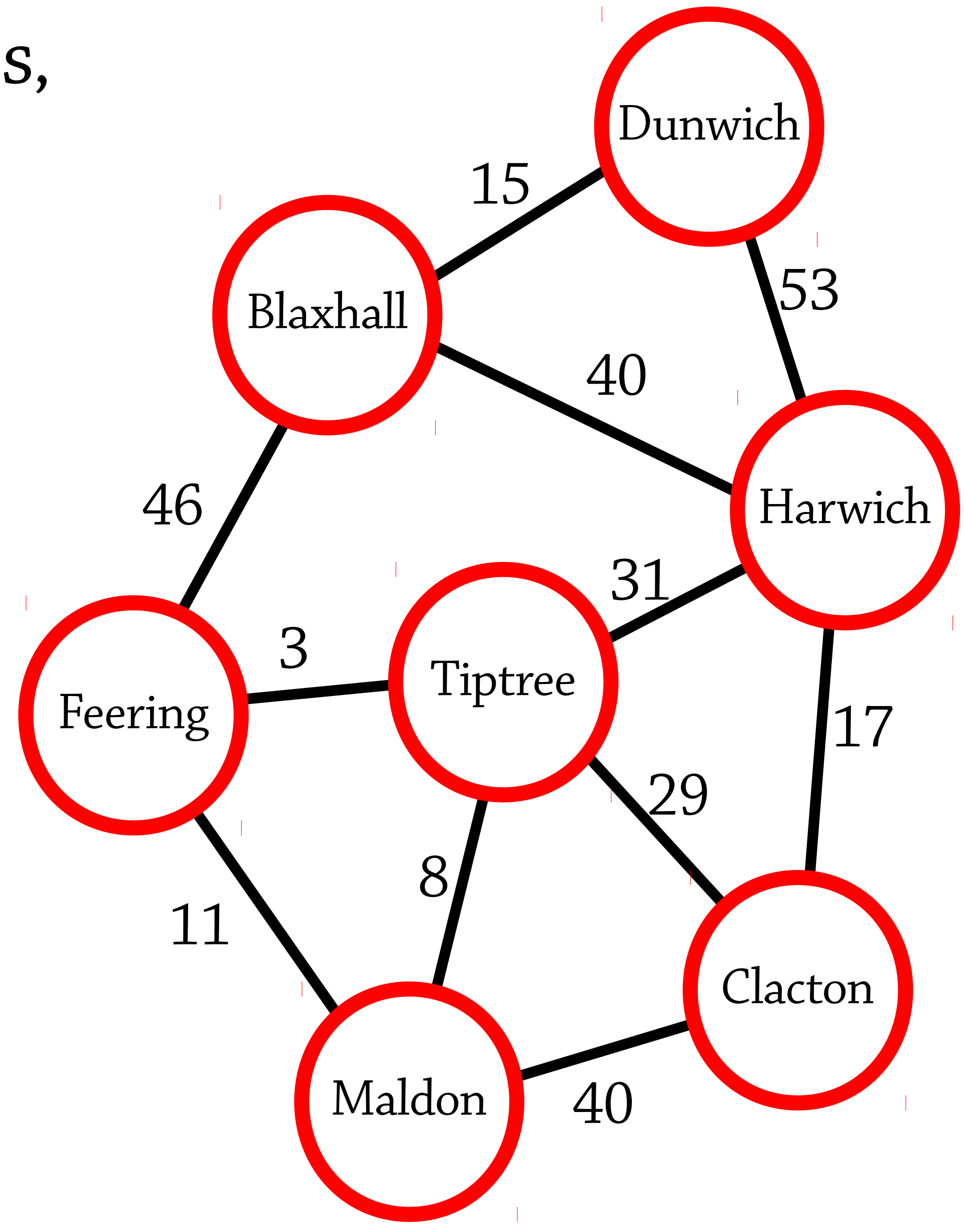
Dijkstra's algorithm

Once we have these distances, we can use them to find the shortest path to any node!

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



hm

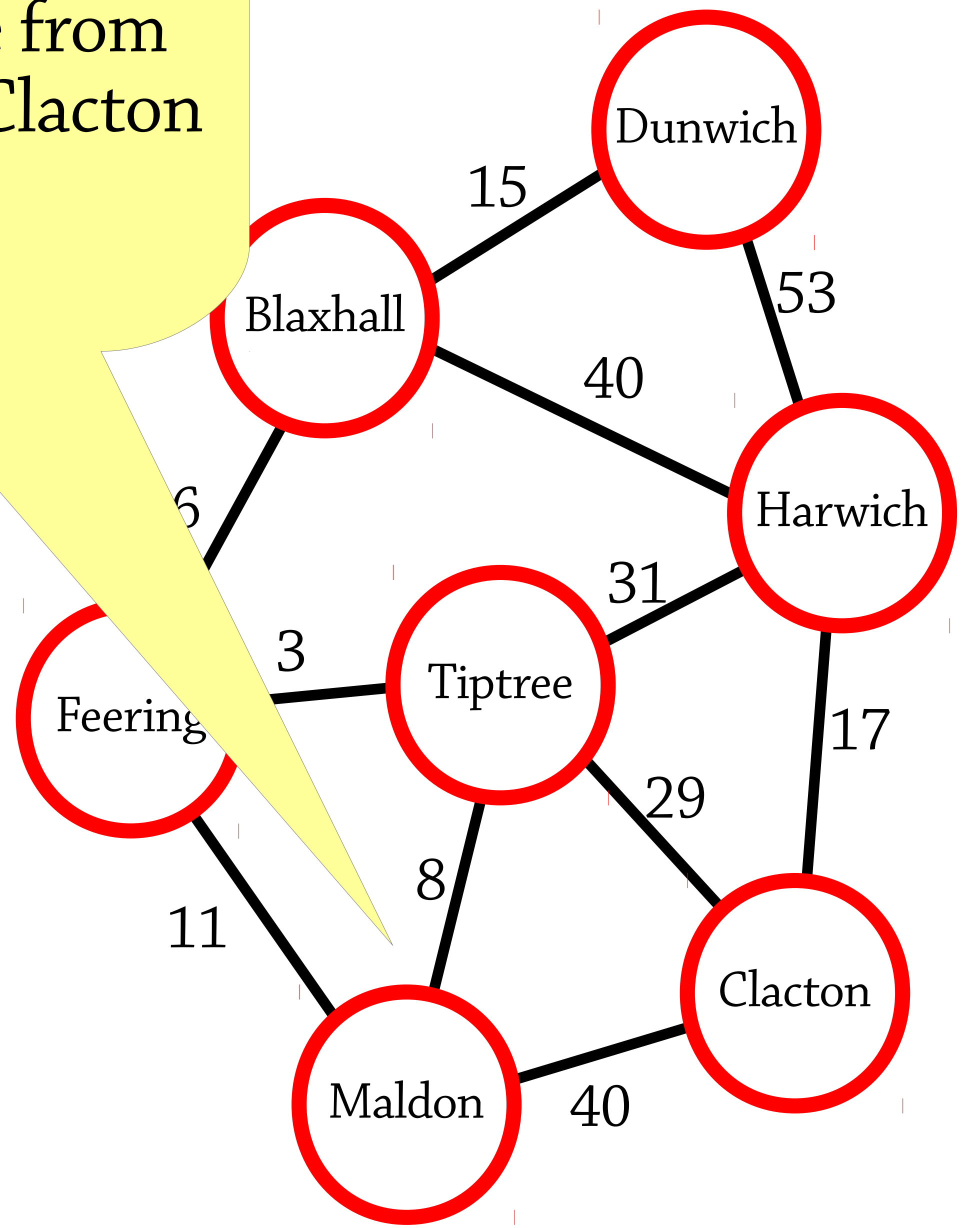
To arrive at Maldon, we must take the edge from Feering, Tiptree or Clacton

Once we
we can us
shortest p

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



m

Dunwich → Clacton: **70**
Clacton → Maldon edge: **40**

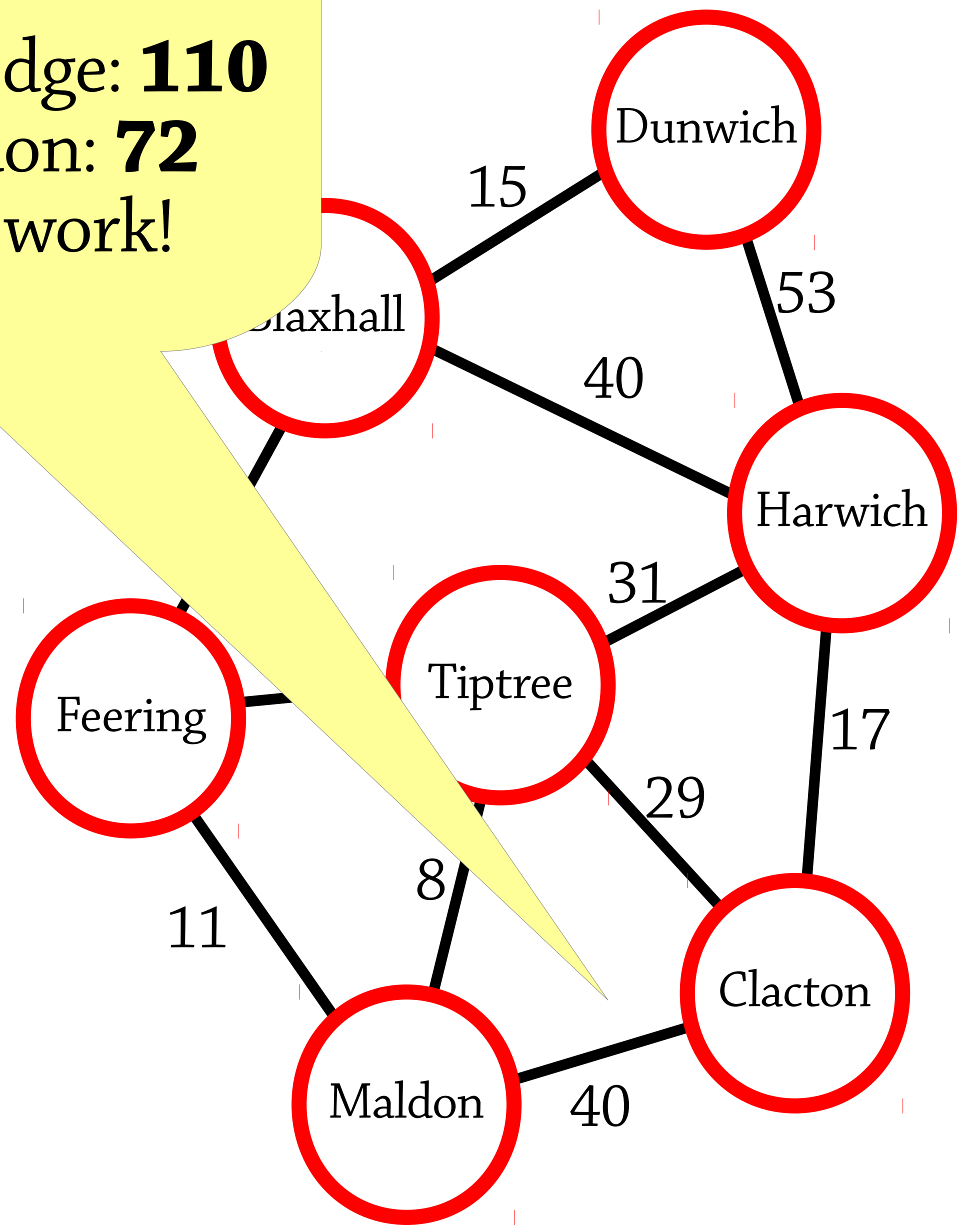
So coming via this edge: **110**
Dunwich → Maldon: **72**
This route won't work!

Once we
we can us
shortest p

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



Dunwich → Tiptree: **64**
Tiptree → Maldon edge: **8**

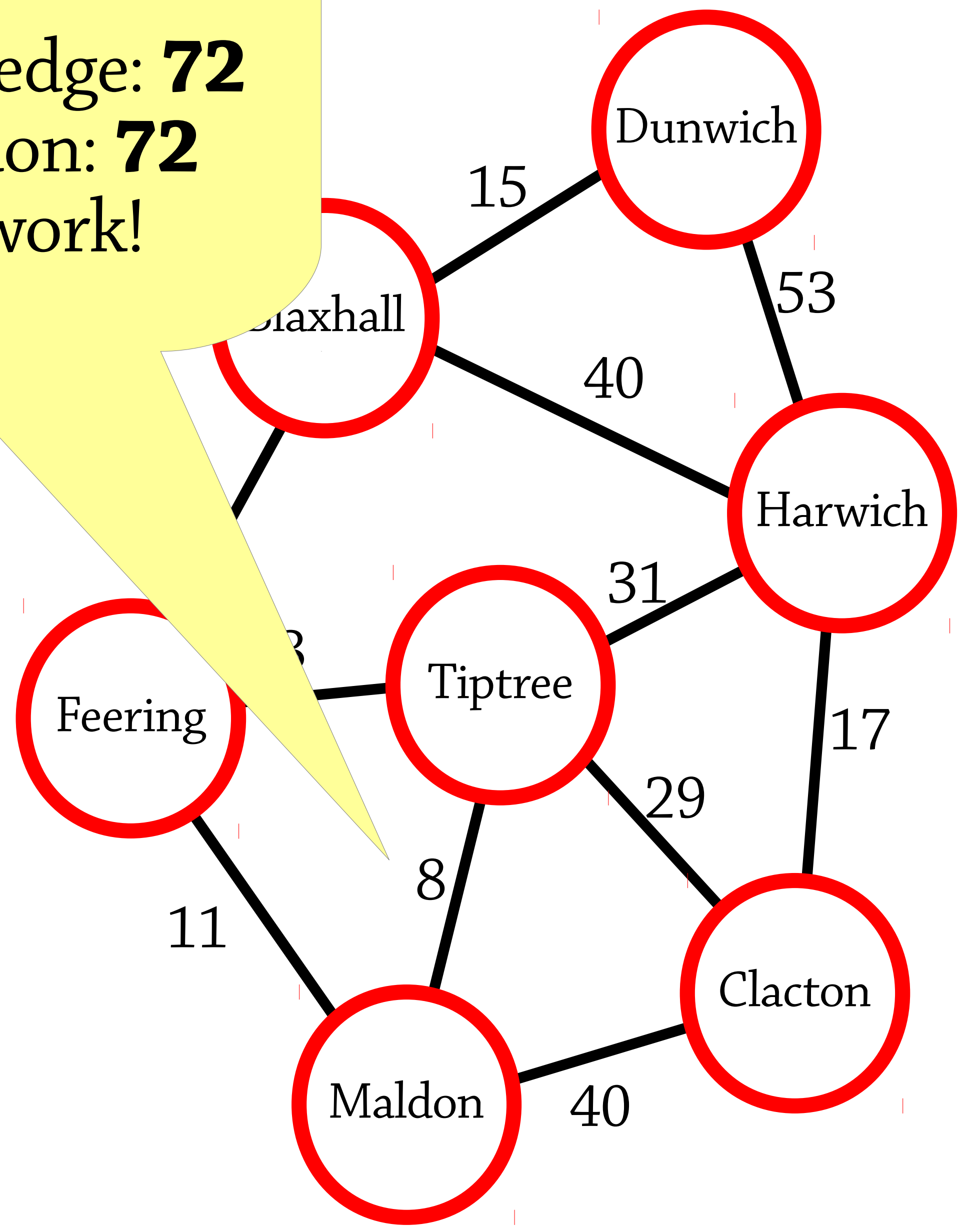
So coming via this edge: **72**
Dunwich → Maldon: **72**
This route will work!

Once we
we can us
shortest p

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



Now we know we can come via Tiptree – so just repeat the process to work out how to get to Tiptree!

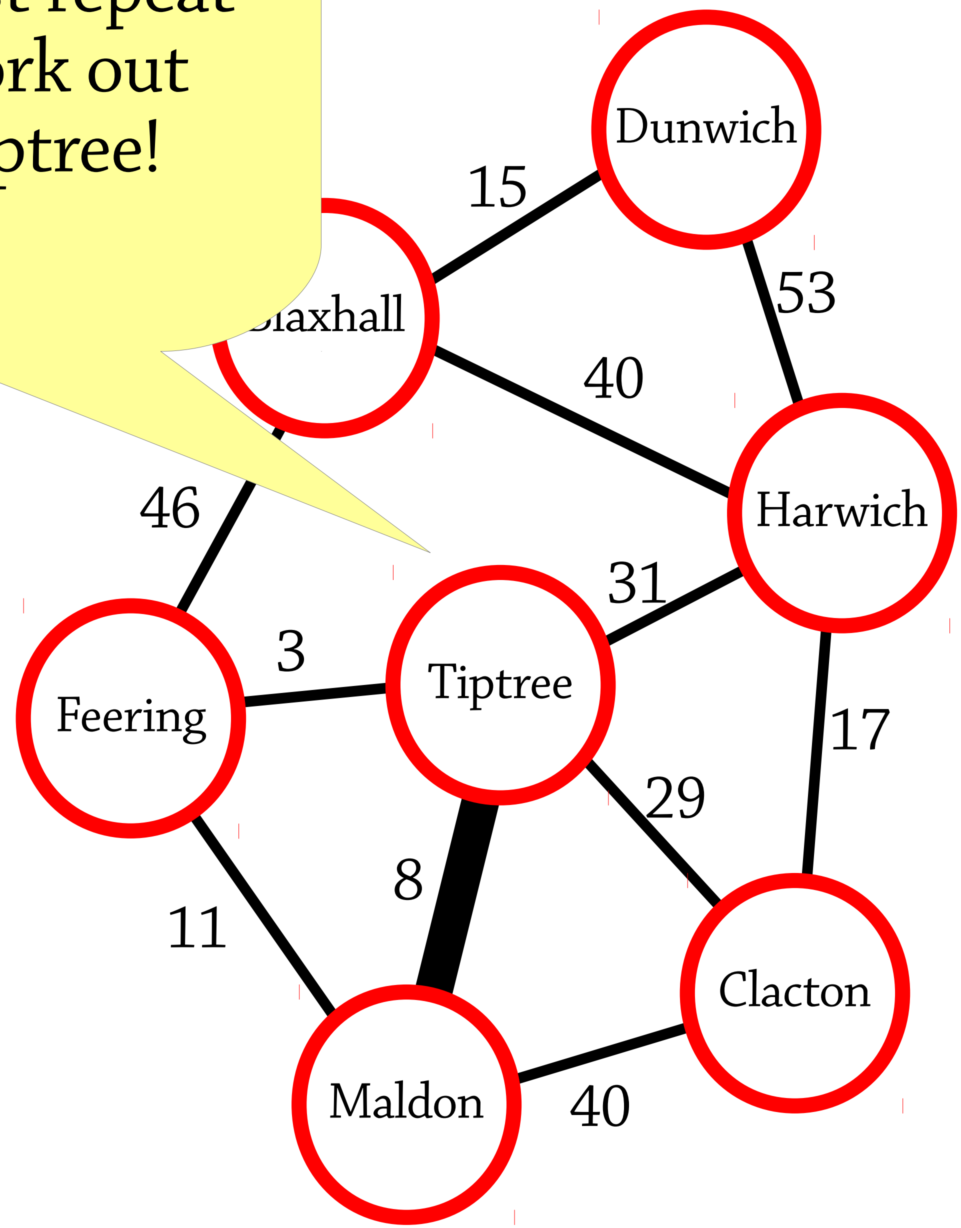
m

Once we
we can us
shortest p

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



Dunwich → Harwich: **53**
Harwich→ Tiptree edge: **31**

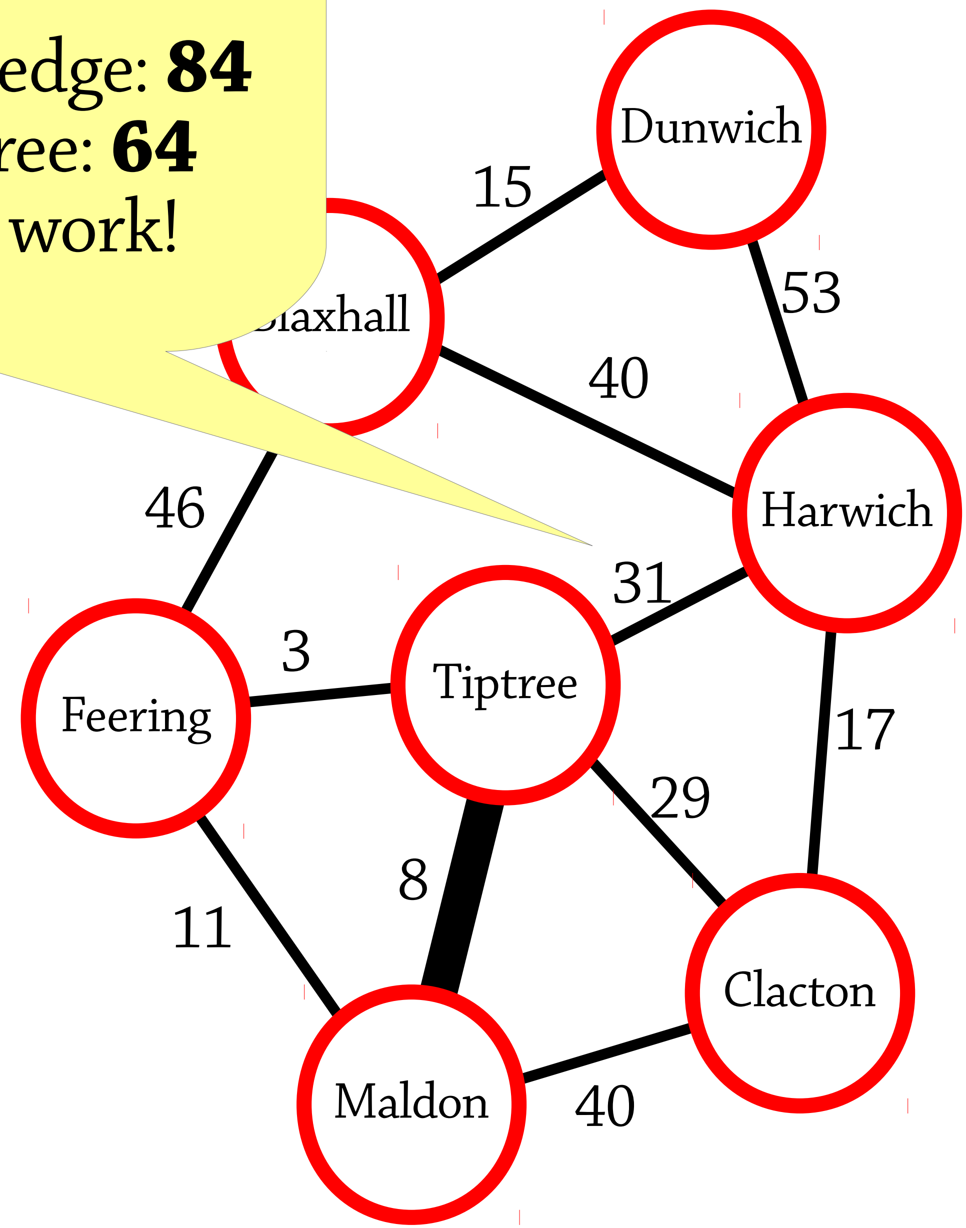
So coming via this edge: **84**
Dunwich → Tiptree: **64**
This route won't work!

Once we
we can us
shortest p

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



Dunwich → Feering: **61**
Feering → Tiptree edge: **3**

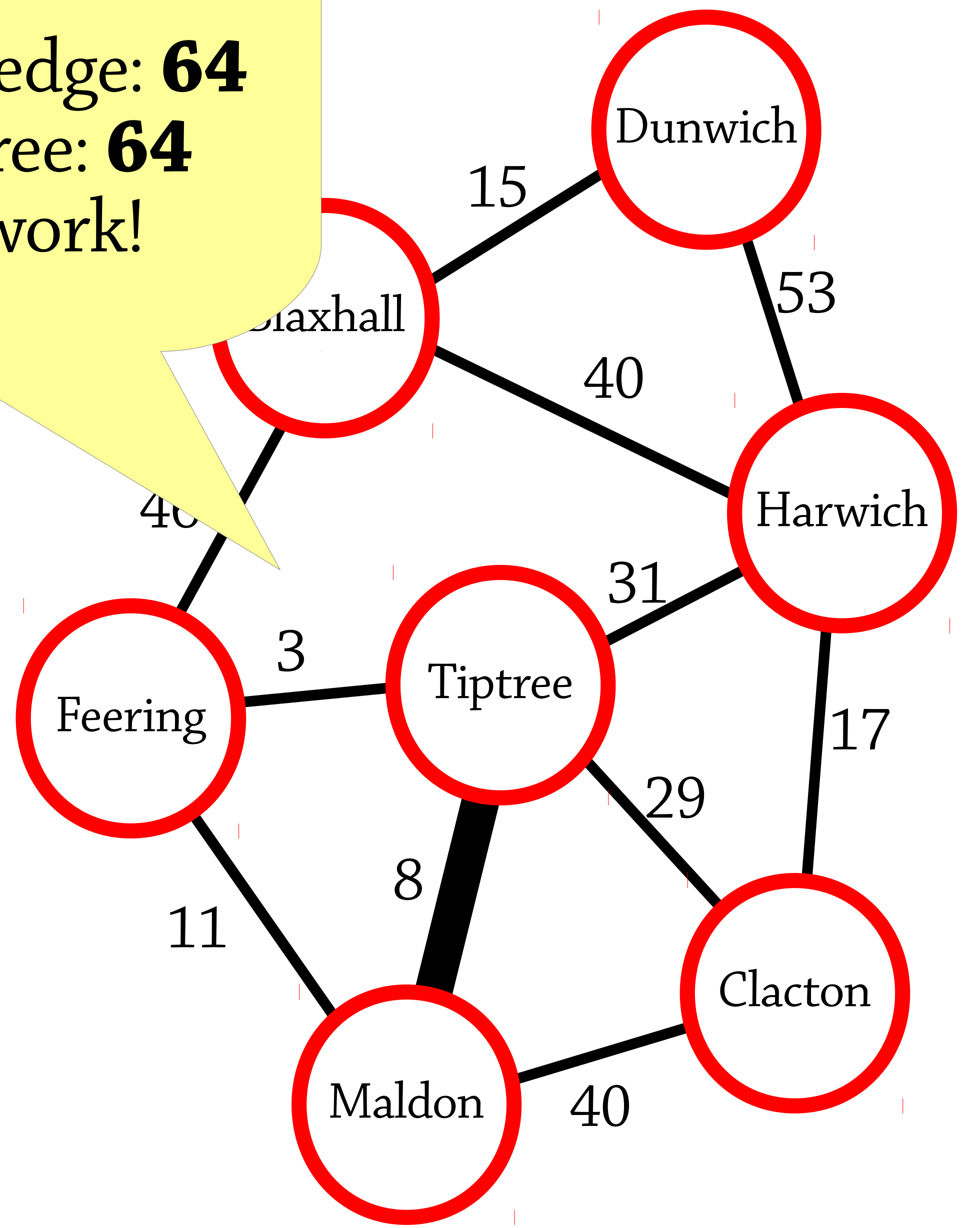
So coming via this edge: **64**
Dunwich → Tiptree: **64**
This route will work!

Once we
we can us
shortest

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



m

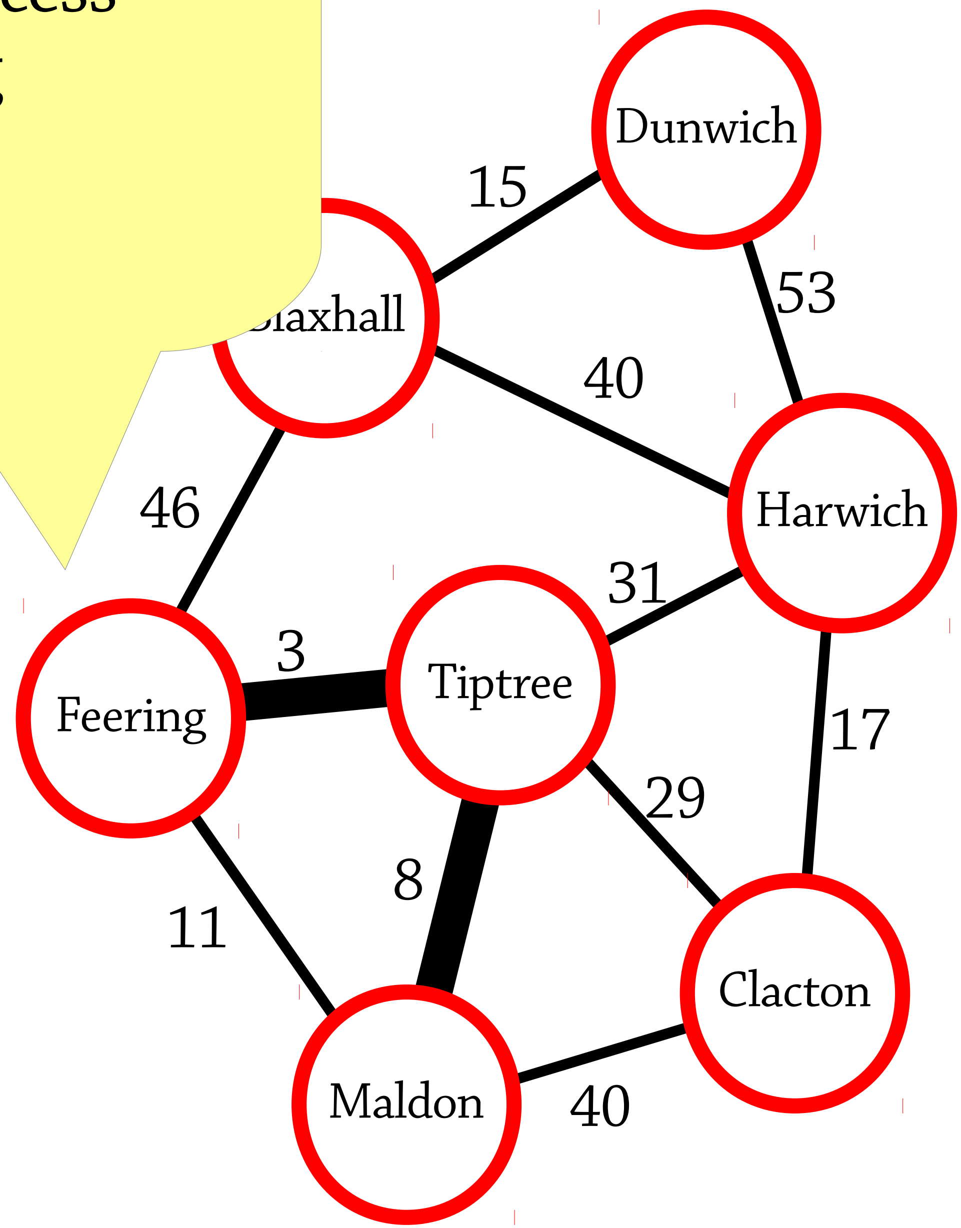
Repeat the process for Feering

Once we
we can us
shortest

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



Dunwich → Blaxhall: **15**
Blaxhall → Feering edge: **46**

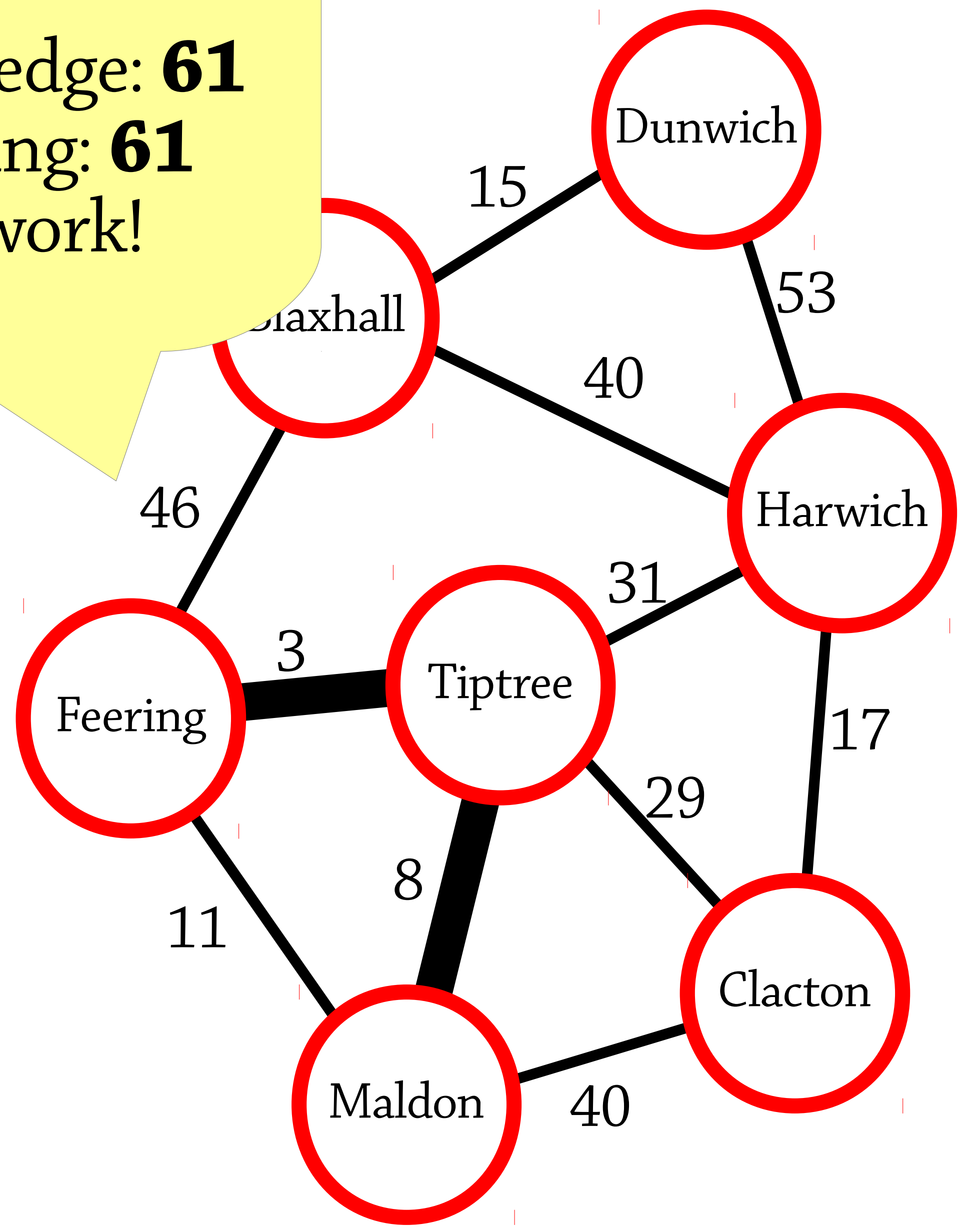
So coming via this edge: **61**
Dunwich → Feering: **61**
This route will work!

Once we
we can us
shortest p

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



algorithm

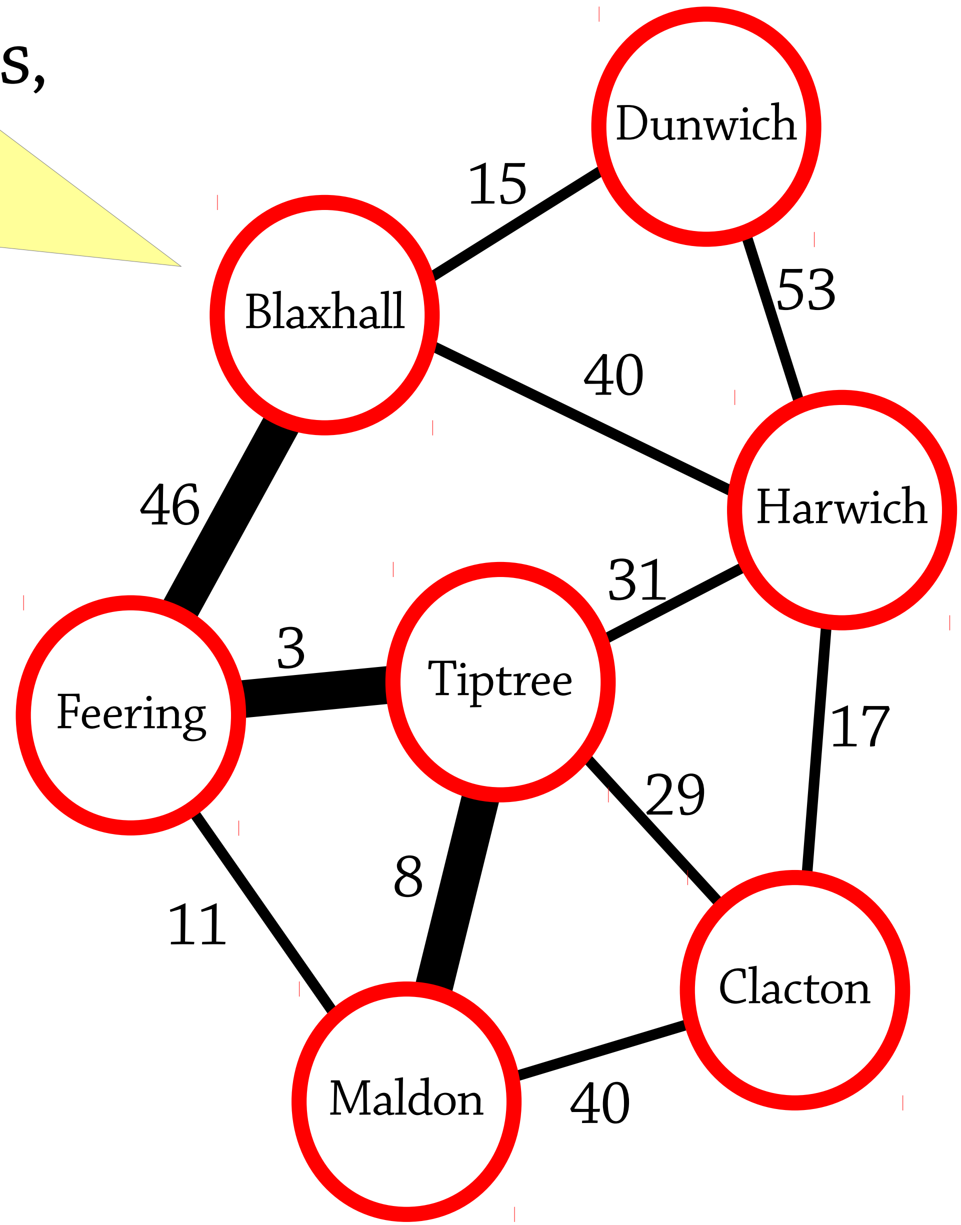
Repeat the process for Blaxhall

ces,

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



Dunwich → Harwich: **53**
Harwich → Blaxhall edge: **40**

So coming via this edge: **93**
Dunwich → Blaxhall: **15**
This route won't work!

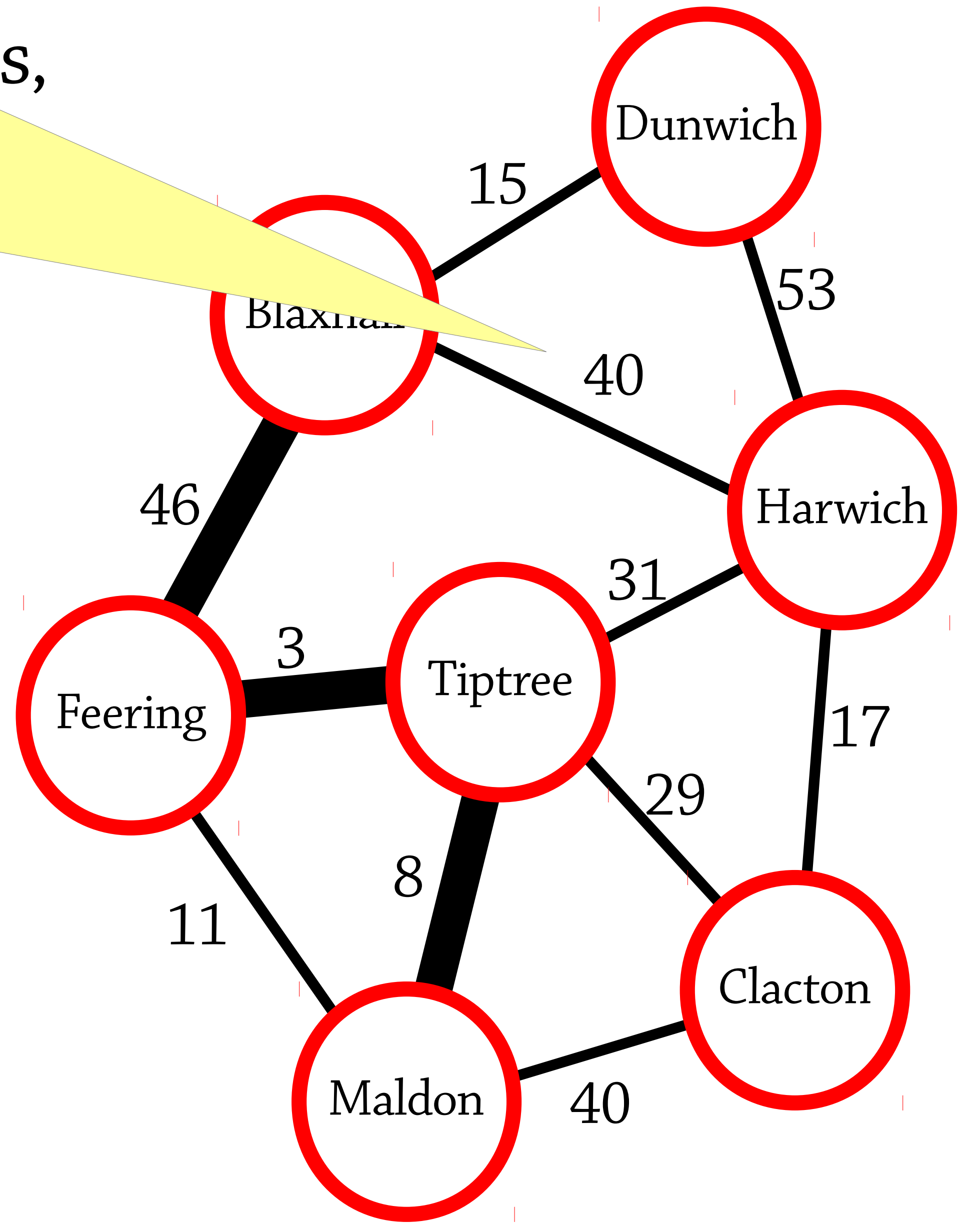
algorithm

ces,

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



…algorithm

Dunwich → Dunwich: **0**
Dunwich → Blaxhall edge: **15**

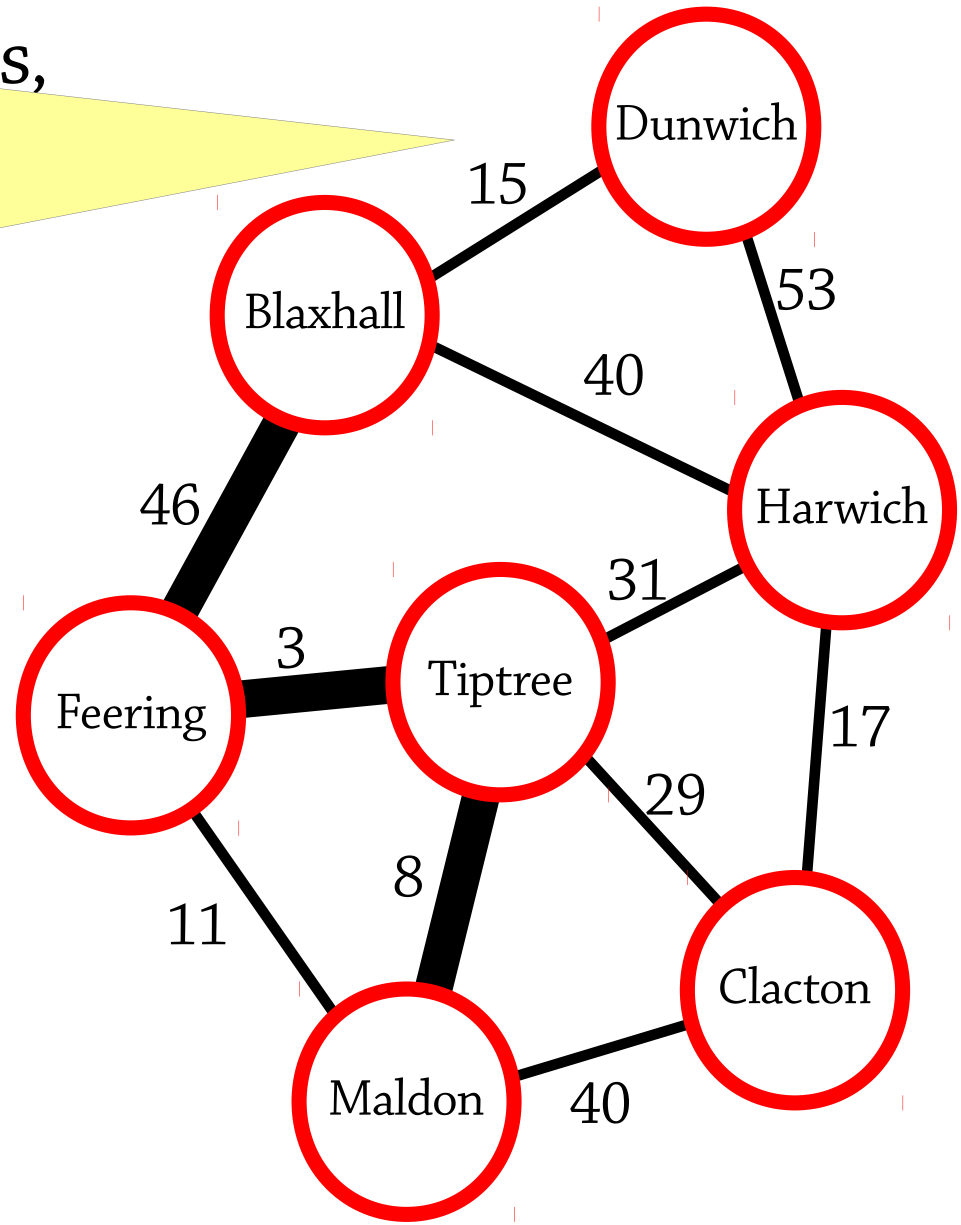
So coming via this edge: **15**
Dunwich → Blaxhall: **15**
This route will work!

…ces,

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



…algorithm

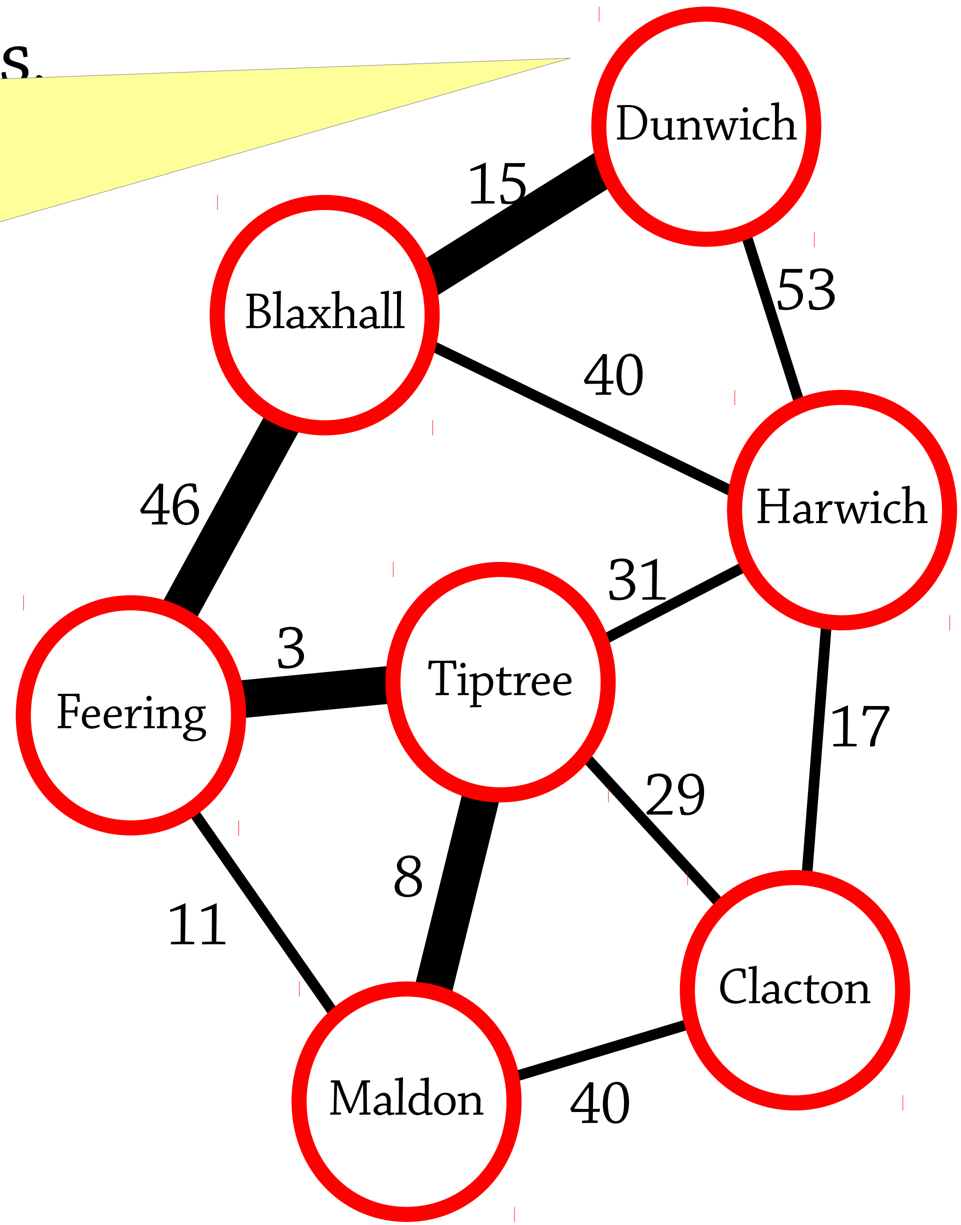
Now we have found our way back to the start node and have the shortest path!

…ces.

e.g. take Maldon

Idea: work out which edge we should take on the final leg of the journey

Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72



Dijkstra's algorithm

Formally, we maintain a set S, which contains all visited nodes and their distances (really a map)

Let S = {start node → 0}
While not all nodes are in S,

- For each node *x* in S, and each neighbour *y* of *x*, calculate
  d = distance to x + cost of edge from x to y
- Find the node *y* which has the smallest value for *d*
- Add that *y* and its distance *d* to S

This computes the shortest distance to each node, from which we can reconstruct the shortest path to any node

What is the efficiency of this algorithm?

Each time through the outer loop, we loop through all edges in S, which by the end contains |E| edges

ra's algori

We add one node to S each time through the loop – loop runs |V| times

n a set S, whicl

ances (really a ma

Let S = {star le → 0}

While not all no es are in S

- For each node x in S, and each neighbour y of x, calculate *d = distance to x + cost of edge from x to y*
- Find the node y which has the smallest r *d*
- Add that y and its distance d to S

This computes the shorte node, from which we can reconstru to any node

What is the efficiency of t

Total: O(|V| × |E|)!

Dijkstra's algorithm, made efficient

The algorithm so far is $O(|V| \times |E|)$

This is because this step:

- For all nodes adjacent to a node in S, calculate their distance from the start node, and pick the closest one

takes $O(|E|)$ time, and we execute it once for every node in the graph

How can we make this faster?

Dijkstra's algorithm, made efficient

Answer: use a priority queue!

To find the closest unvisited node, we store all *neighbours* of unvisited nodes in a priority queue, together with their distances

Instead of searching for the nearest unvisited node, we can just ask the priority queue for the node with the smallest distance

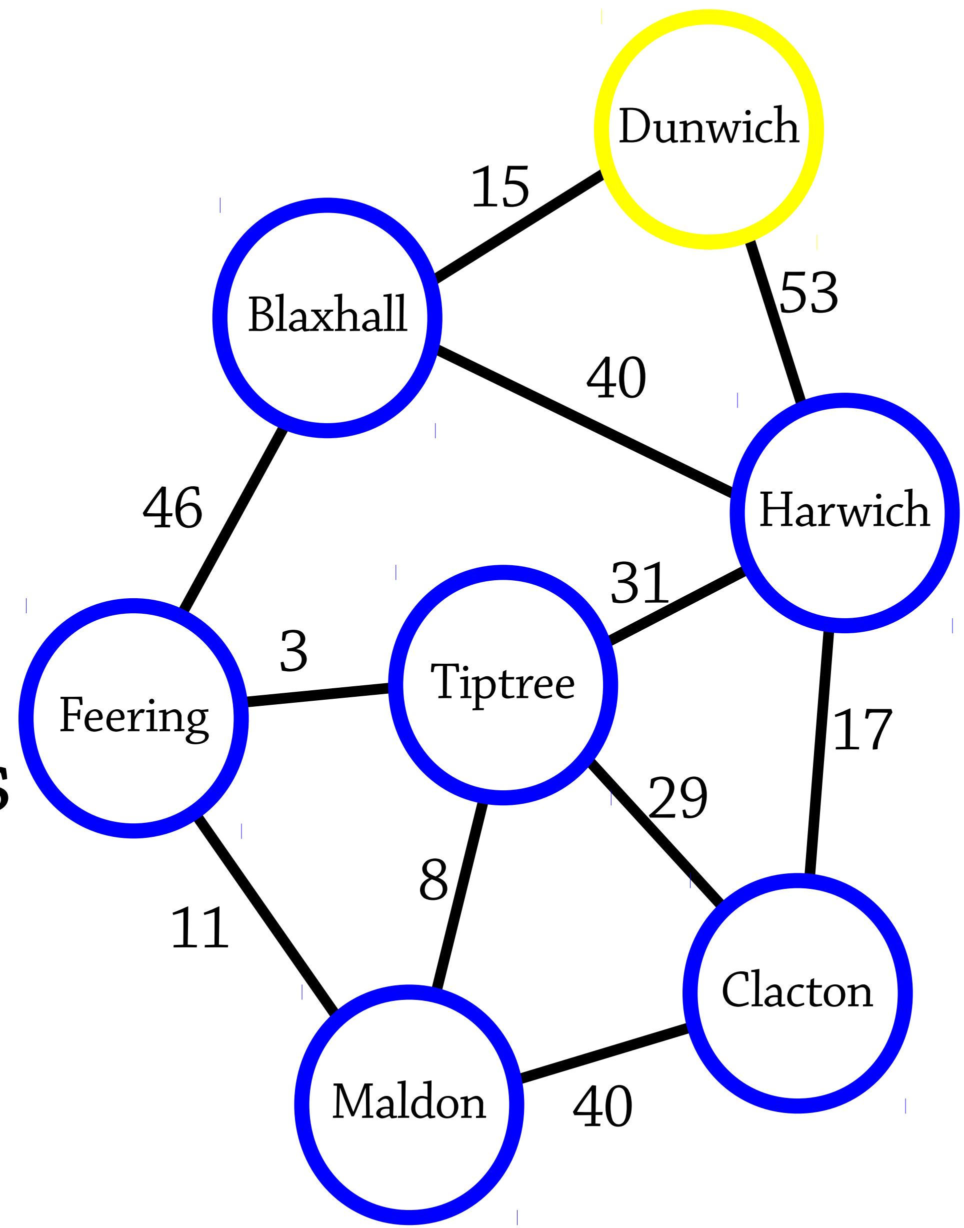
Whenever we visit a node, we will add each of its unvisited neighbours to the priority queue

Dijkstra's algorithm

S = {}

Q = {Dunwich 0},

Remove the smallest element of Q, “Dunwich 0”. Add Dunwich → 0 to S, and add Dunwich's neighbours to Q.

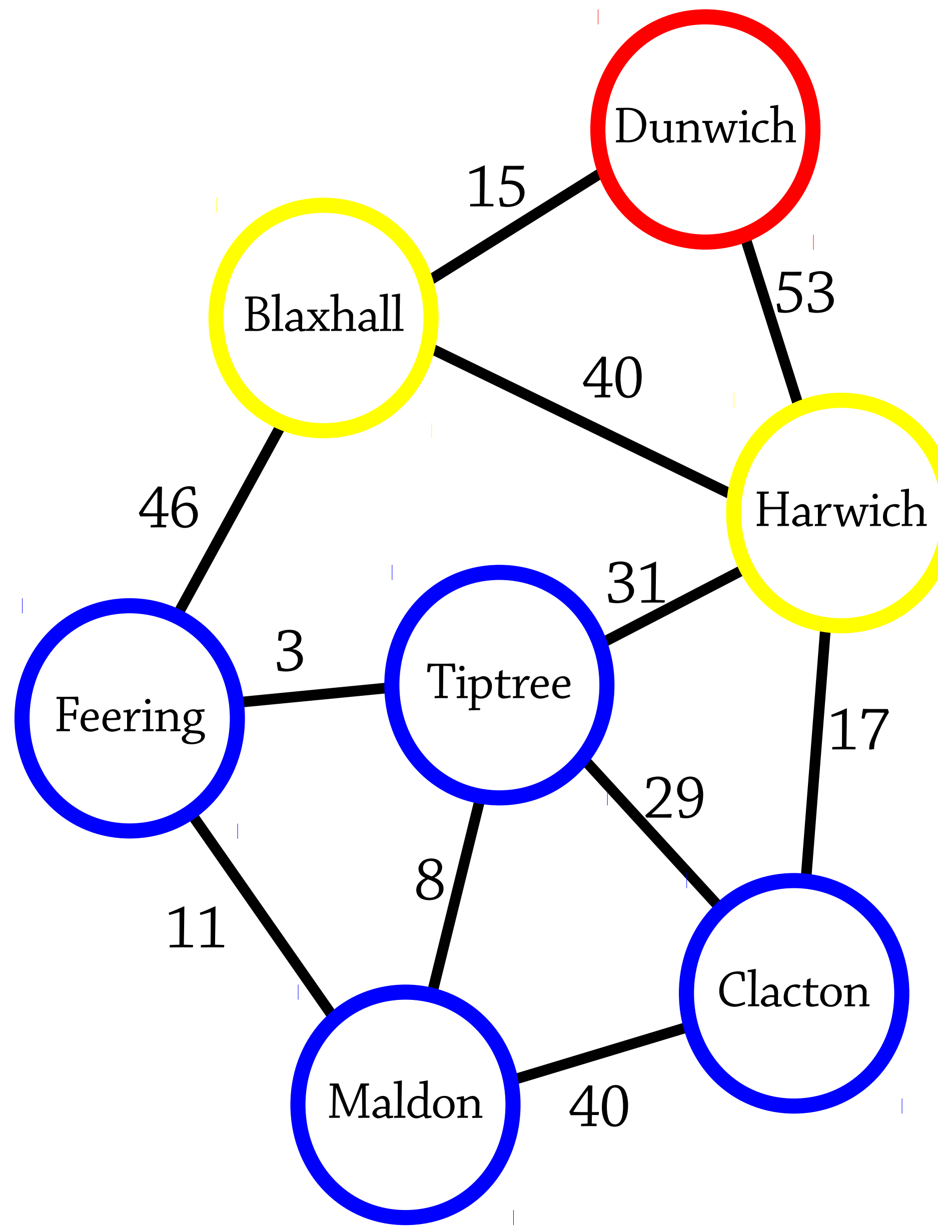


Dijkstra's algorithm

S = {Dunwich → 0}

Q = {Blaxhall 15,
Harwich 53}

Remove the smallest element of Q, "Blaxhall 15". Add Blaxhall → 15 to S, and add Blaxhall's neighbours to Q.

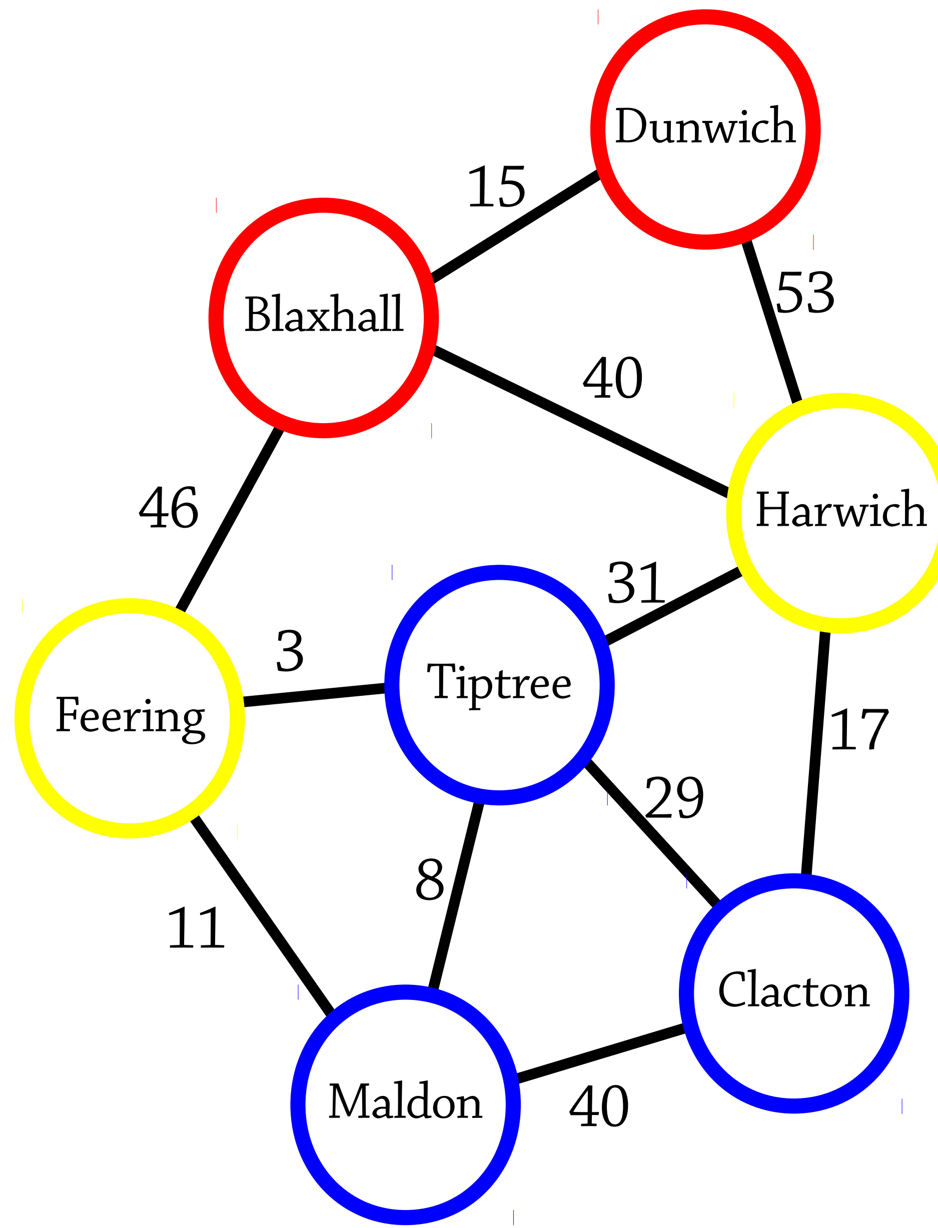


Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15}

Q = {Harwich 53,
Feering 61,
Harwich 55}

Remove the smallest element of Q, “Harwich 53”.
Add Harwich → 53 to S, and add Harwich's neighbours to Q.

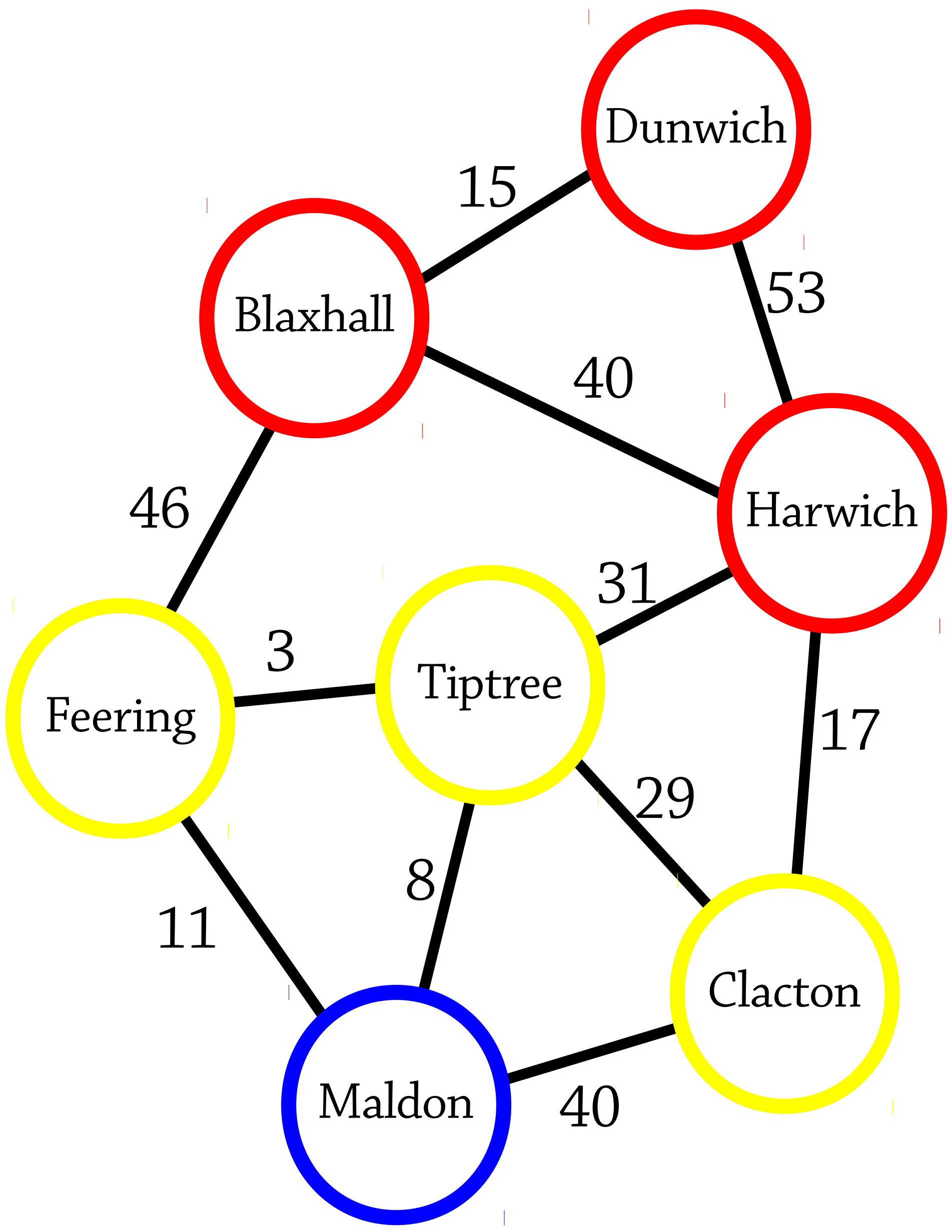


Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15,
Harwich → 53}

Q = {Feering 61,
Harwich 55,
Tiptree 84,
Clacton 70}

Remove the smallest element of Q, "Harwich 55".
Oh! Harwich is already in S.
So just ignore it.

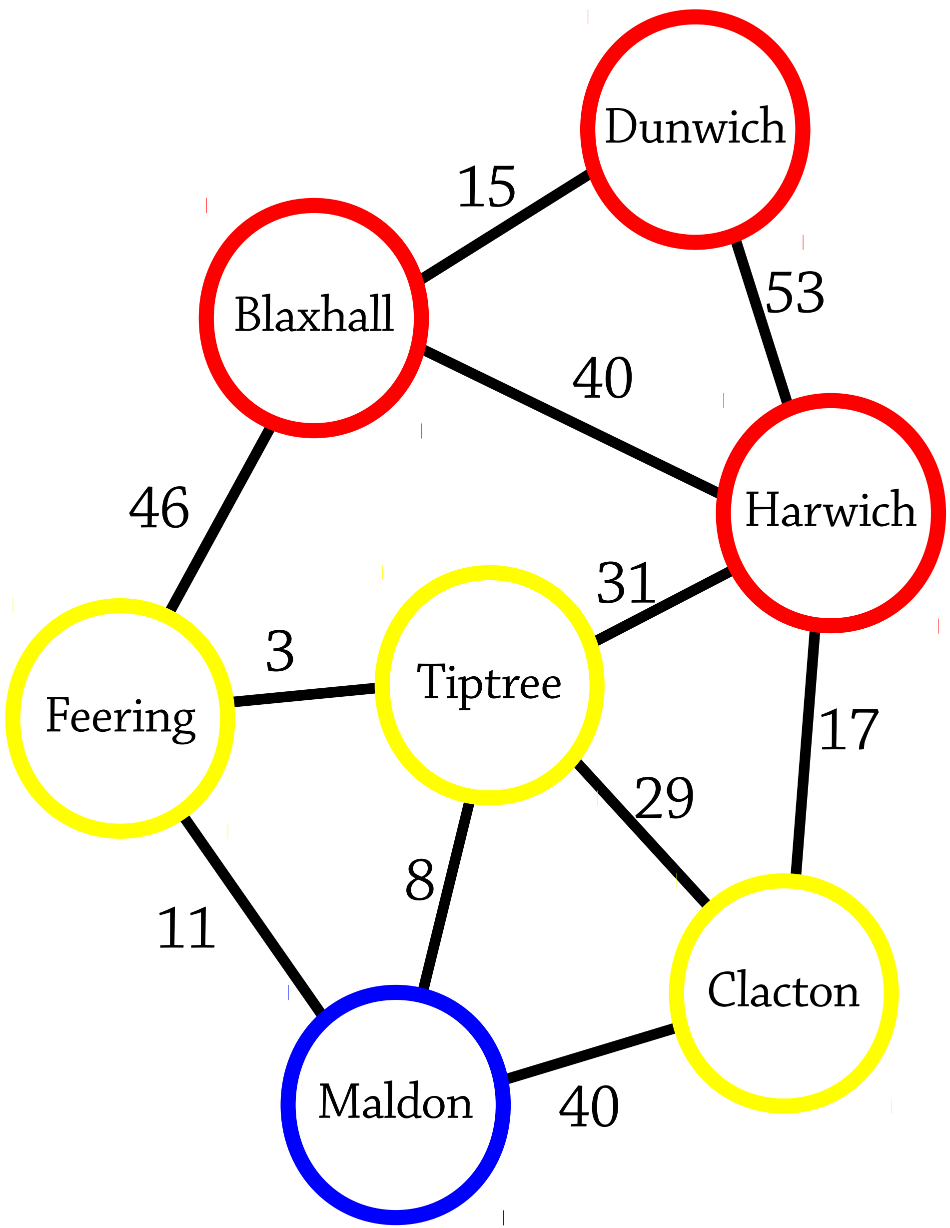


Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15,
Harwich → 53}

Q = {Feering 61,
Tiptree 84,
Clacton 70}

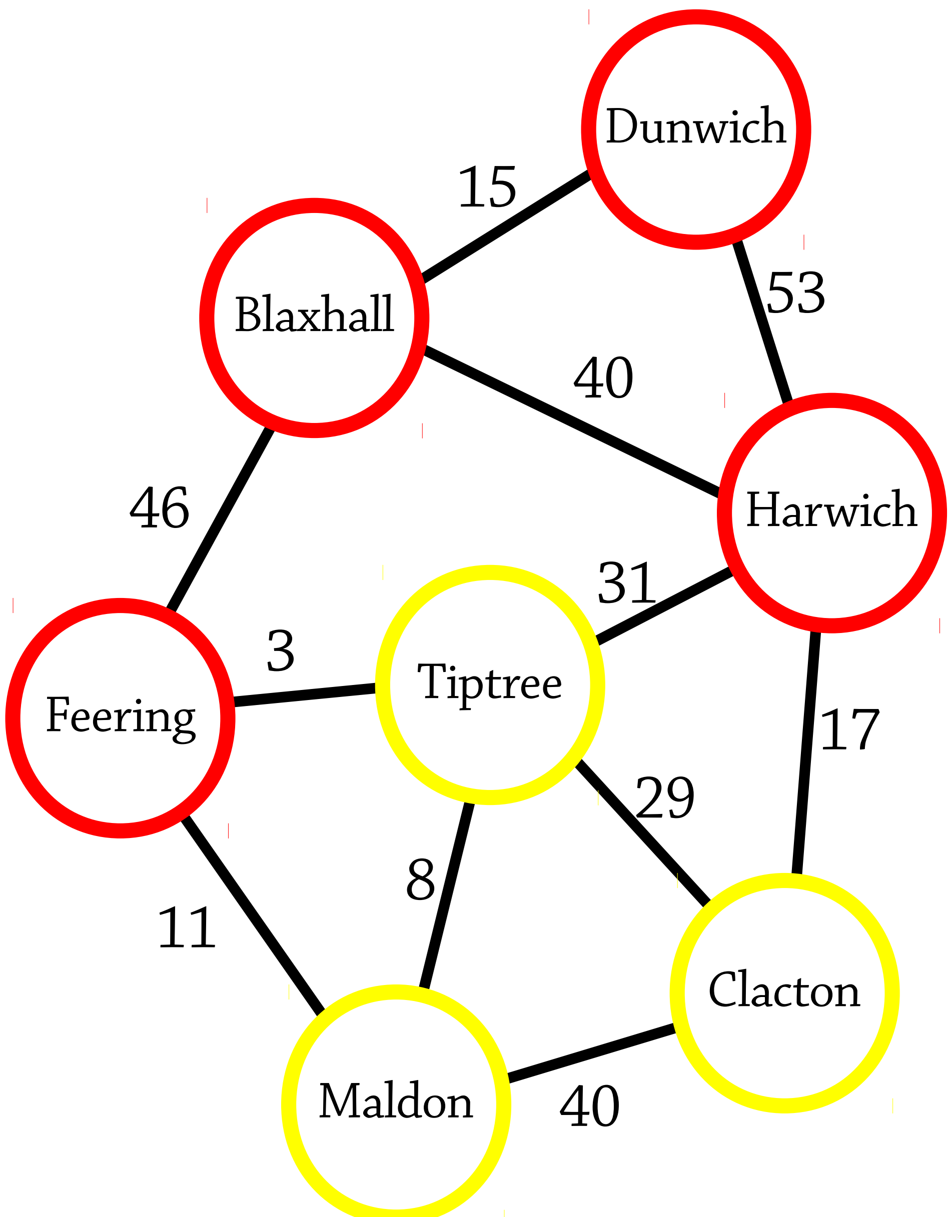
Remove the smallest element of Q, “Feering 61”.
Add Feering → 61 to S, and add Feering's neighbours to Q.



Dijkstra's algorithm

S = {Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61}

Q = {Tiptree 84,
Tiptree 64,
Maldon 72,
Clacton 70}



Dijkstra's algorithm, efficiently

Let S = {} and Q = {start node → 0}

While Q is not empty:

- Remove the node *x* from Q that has the smallest priority (distance), call its distance *d*
- If *x* is in S, do nothing
- Otherwise, add *x* → *d* to S and for each outgoing edge *x* → *y*, add y to Q with priority *d* + *weight of edge to y*

Dij... ...iently

Let S = ...

While ...

Maximum size of Q is $|E|$,
total of $O(|V| + |E|)$
priority queue operations,
so total time:
$O((|V| + |E|) \log |E|)$
or
O(n log n) where $n = |V| + |E|$

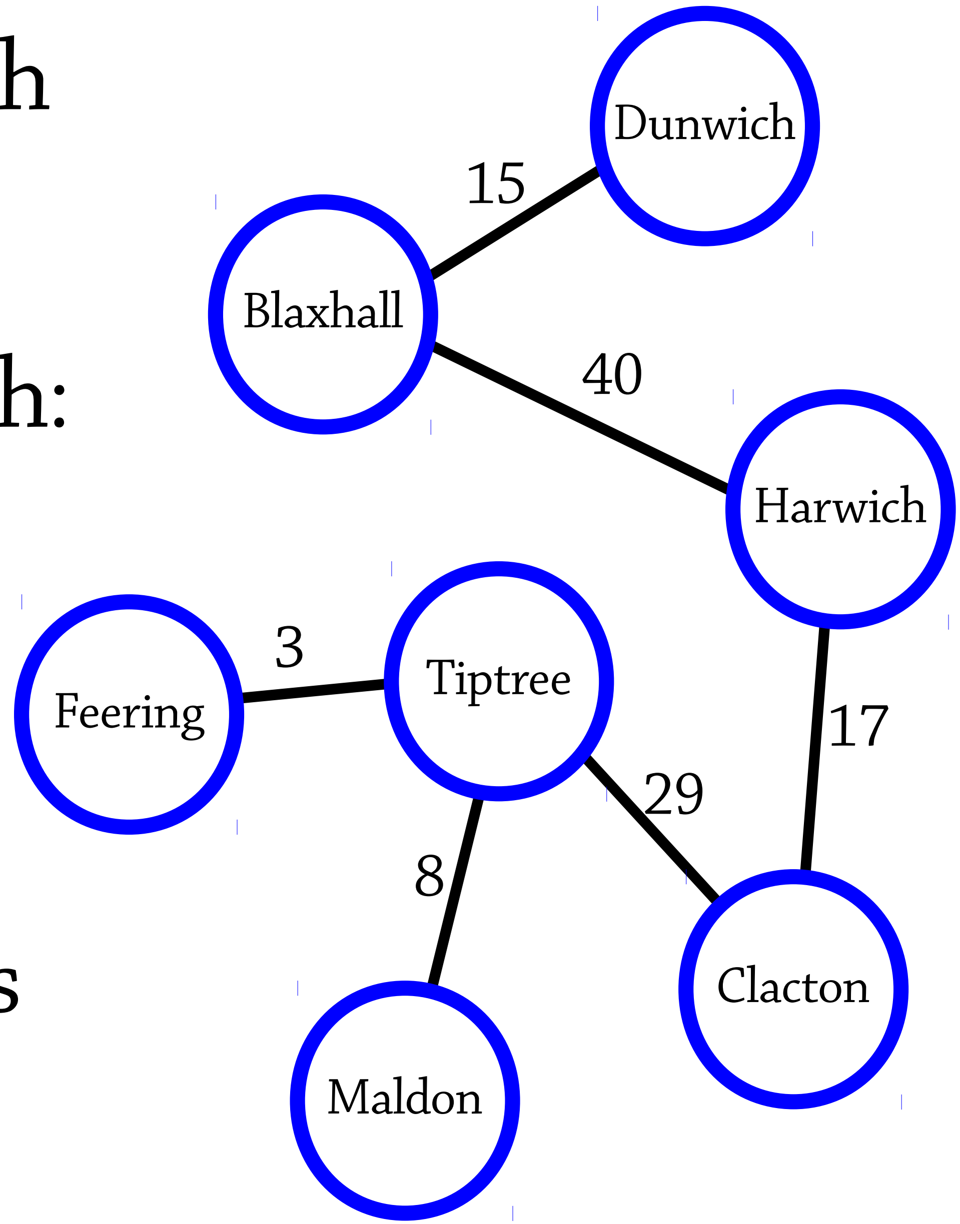
- Remo... ...smallest priority (distance), call its distance *d*
- If *x* is in S, do nothing
- Otherwise, add $x \rightarrow d$ to S and for each outgoing edge $x \rightarrow y$, add y to Q with priority d + *weight of edge to y*

Minimum spanning trees

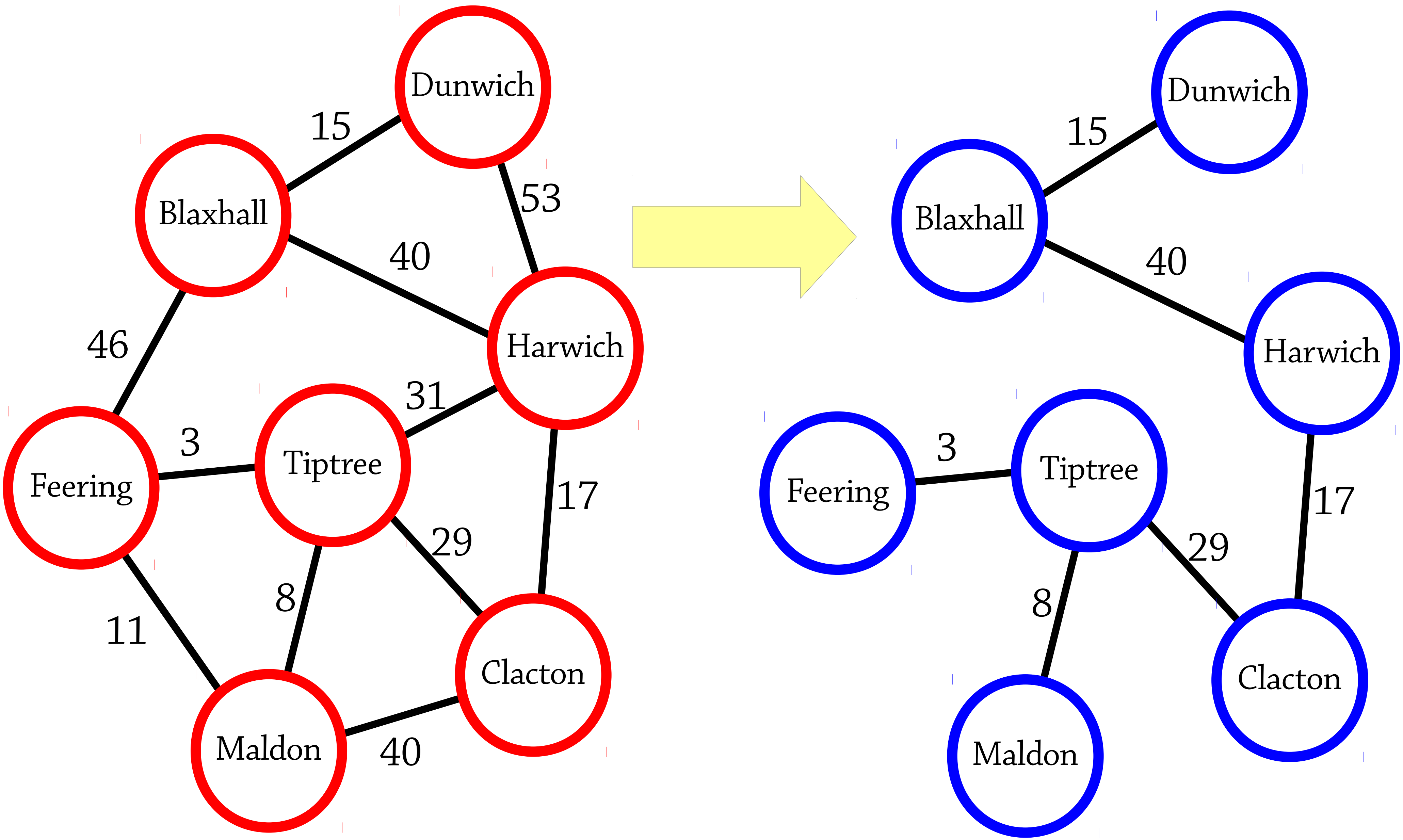
A *spanning tree* of a graph is a subgraph (a graph obtained by deleting some of the edges) which:

- is acyclic
- is connected

A *minimum* spanning tree is one where the total weight of the edges is as low as possible



Minimum spanning trees



Prim's algorithm

We will build a minimum spanning tree by starting with no edges and adding edges until the graph is connected

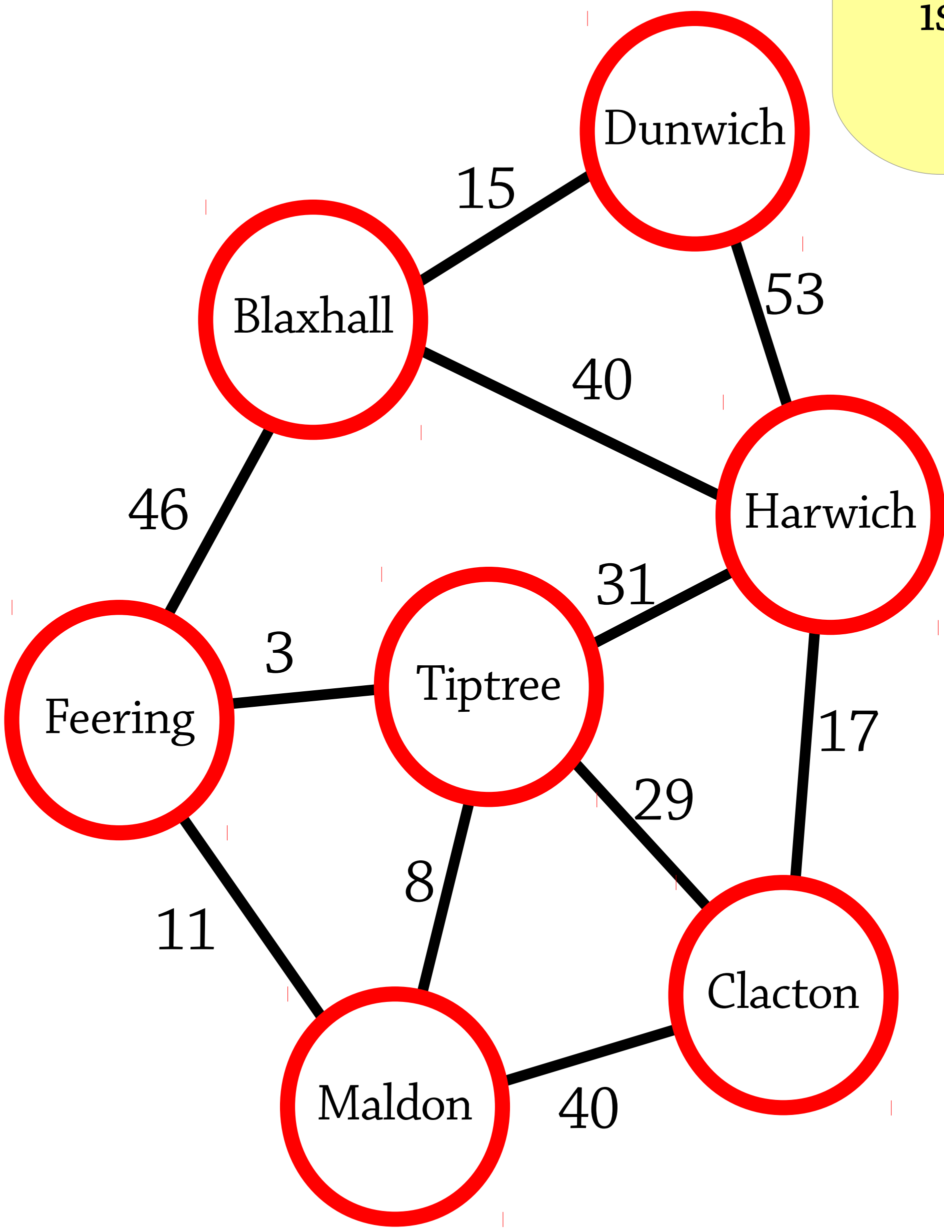
Keep a set S of all the nodes that are in the tree so far, initially containing one arbitrary node

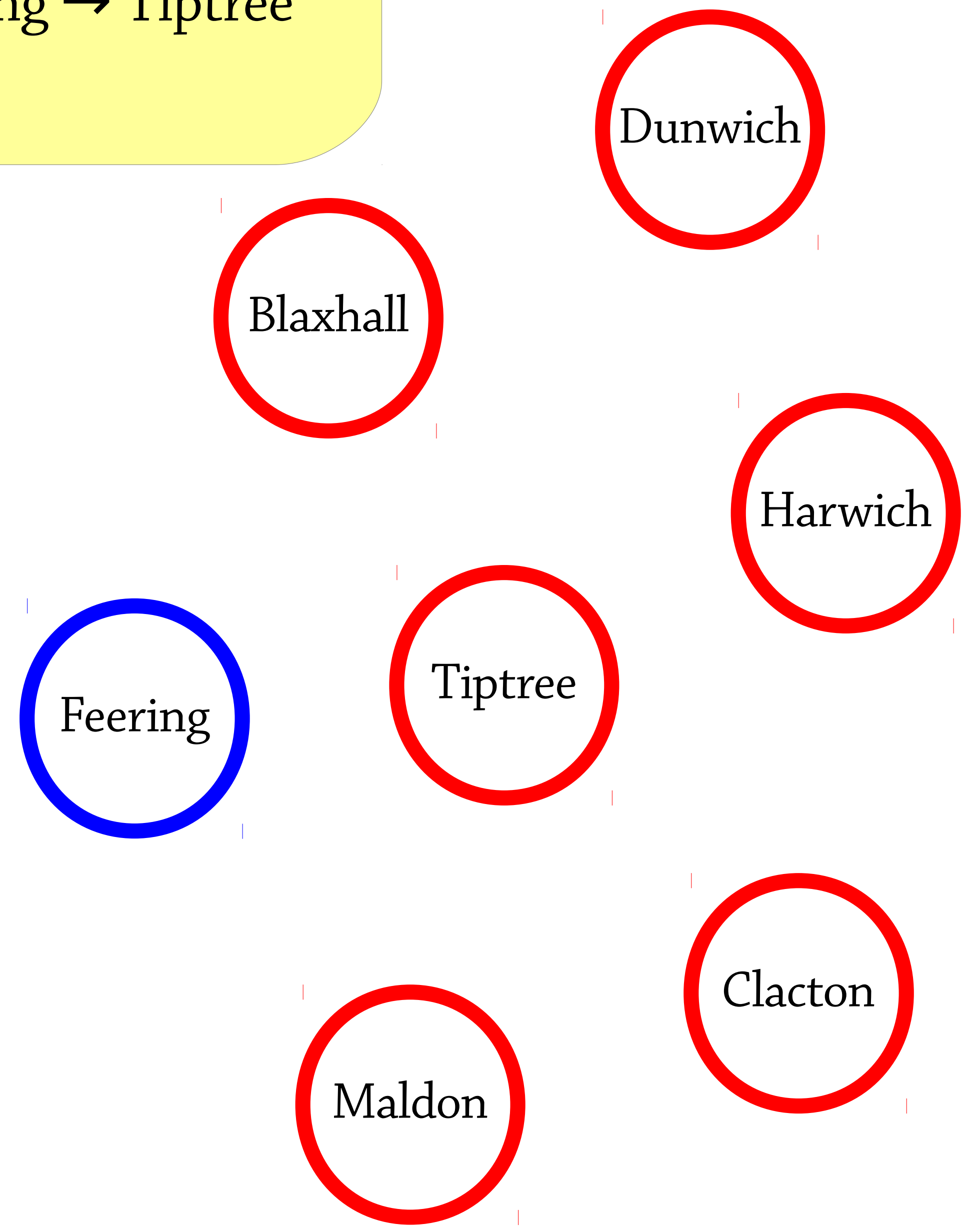
While there is a node not in S:

- Pick the *lowest-weight* edge between a node in S and a node not in S
- Add that edge to the spanning tree, and add the node to S

Minimum ... ees

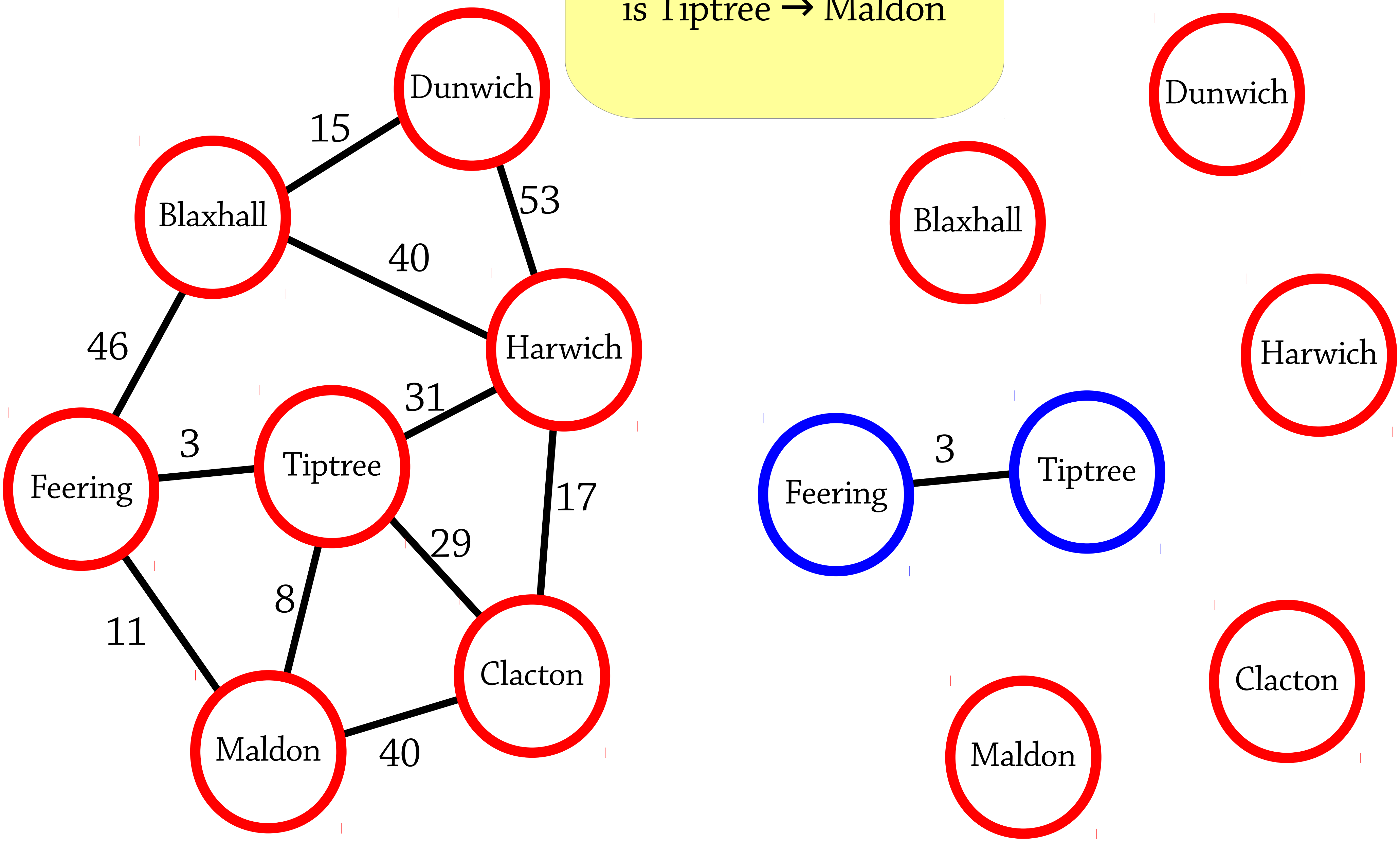
S = {Feering}
Lowest-weight edge
from S to not-S
is Feering → Tiptree





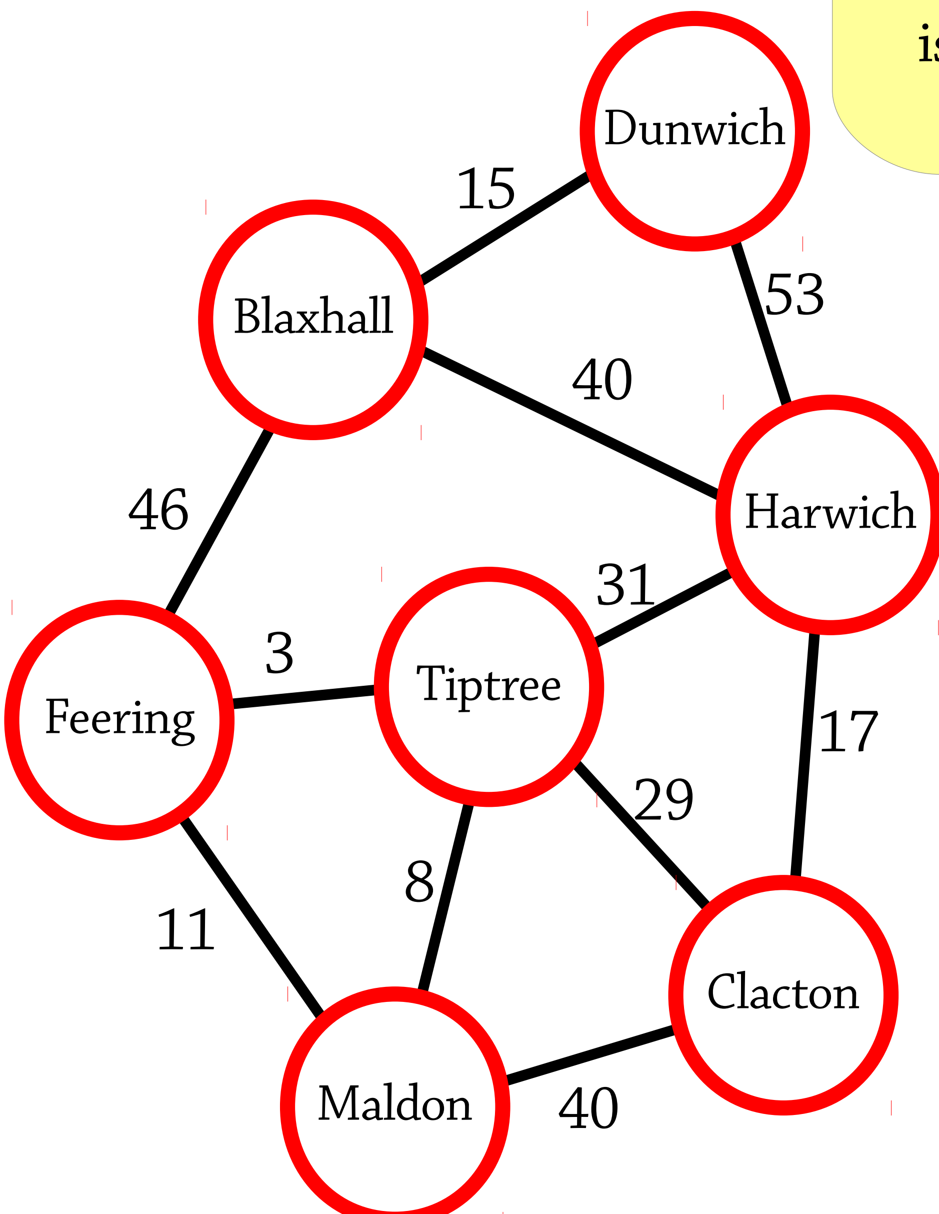
Minimun ees

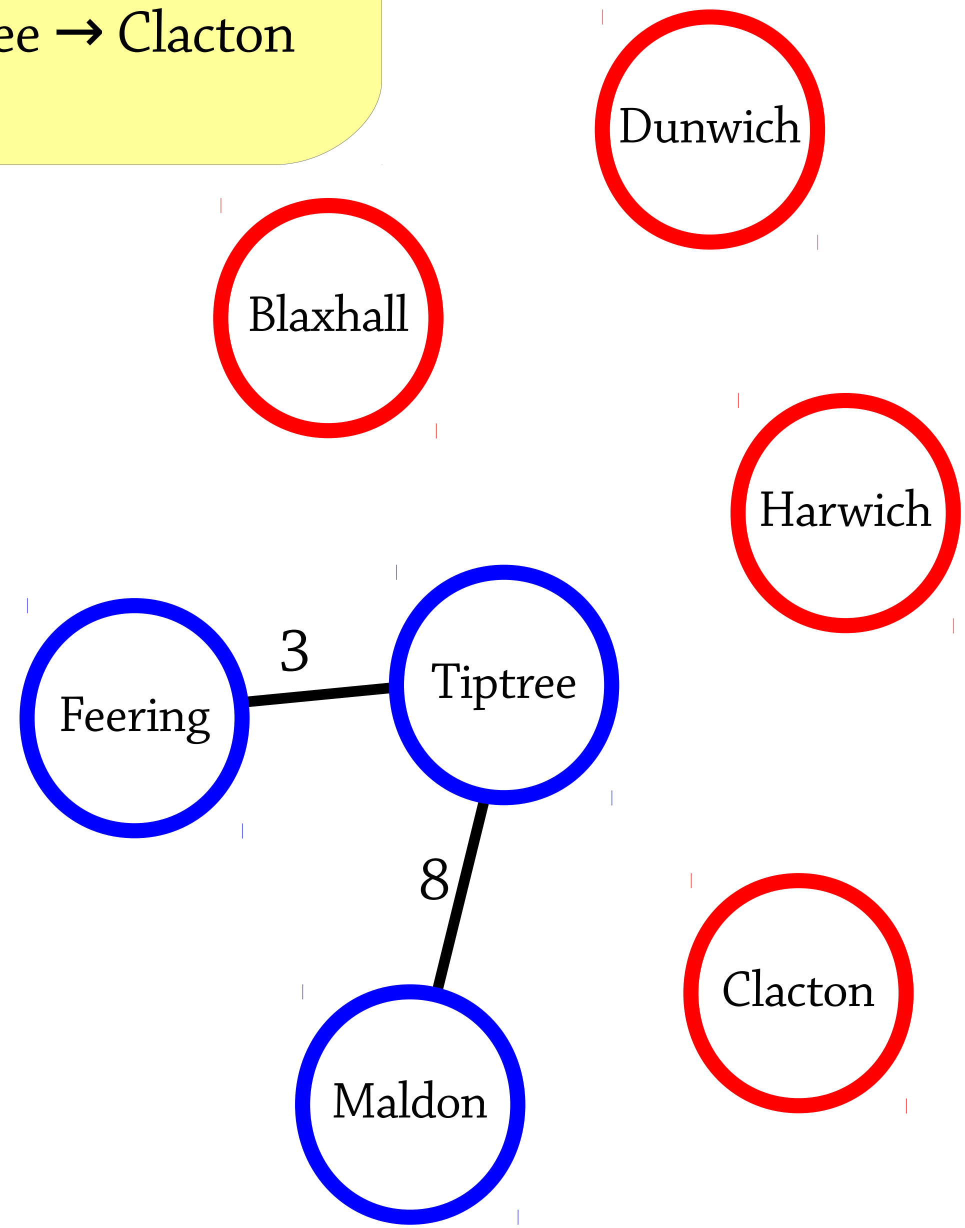
S = {Feering, Tiptree}
Lowest-weight edge
from S to not-S
is Tiptree → Maldon



Minimun ... ees

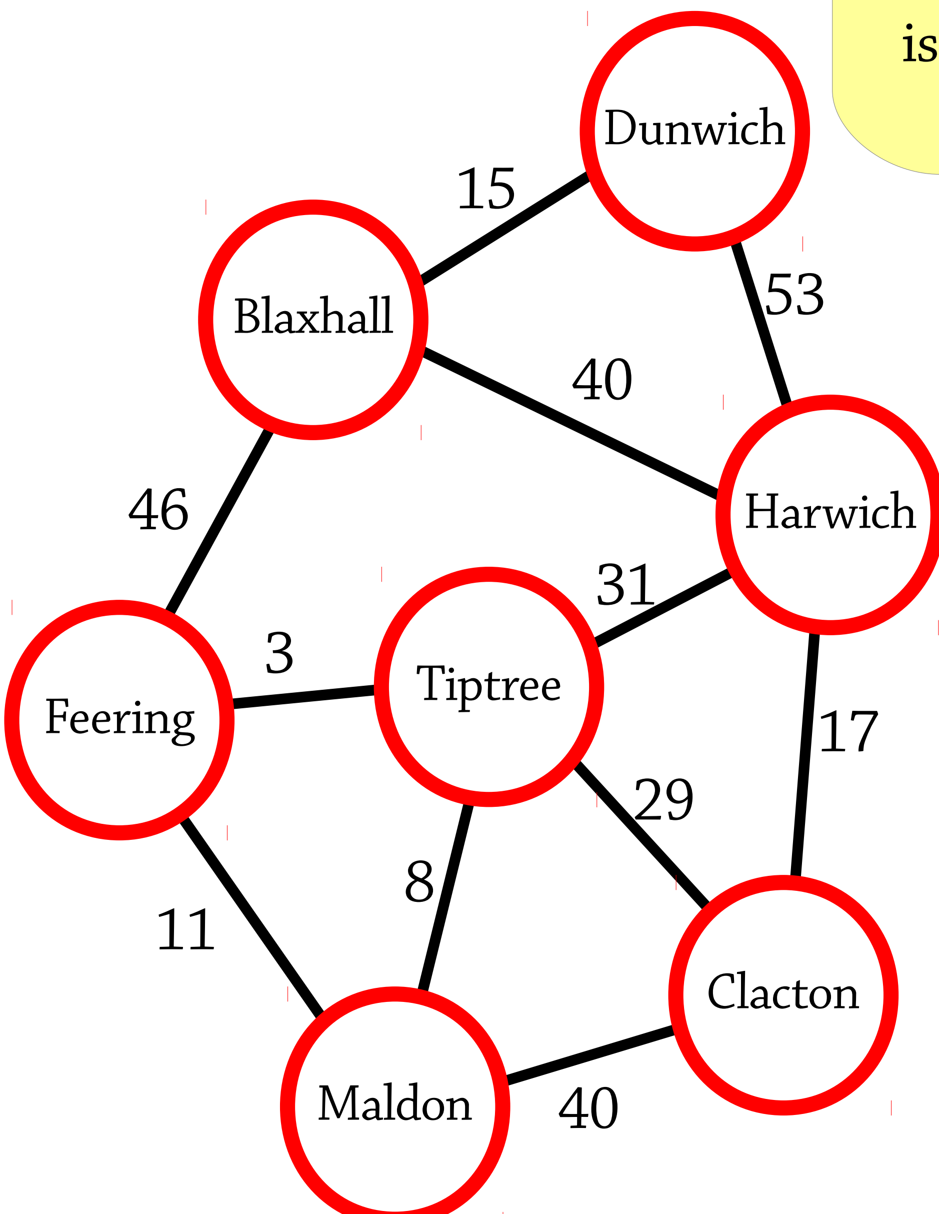
S = {Feering, Tiptree, Maldon}
Lowest-weight edge from S to not-S is Tiptree → Clacton

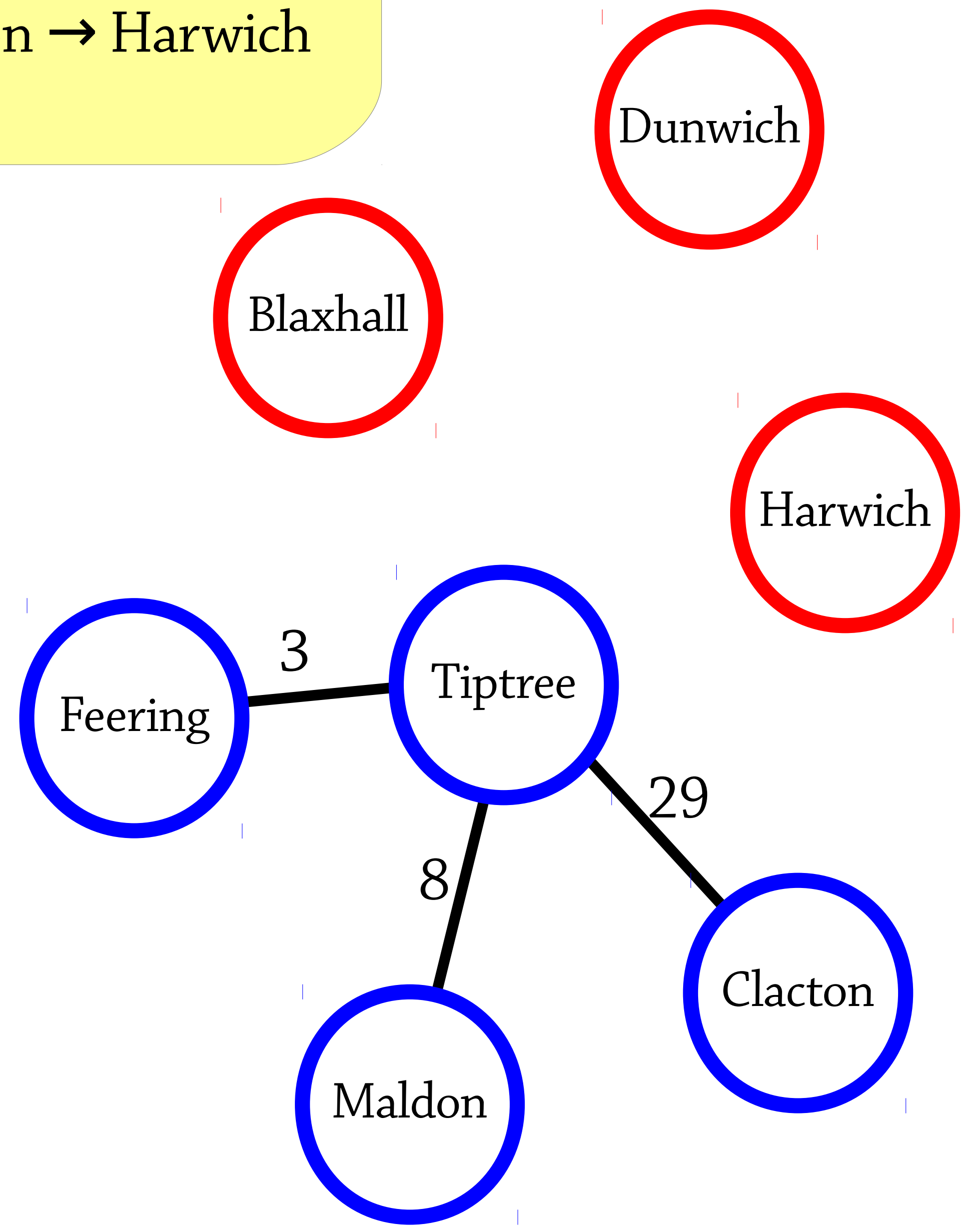




Minimun … ees

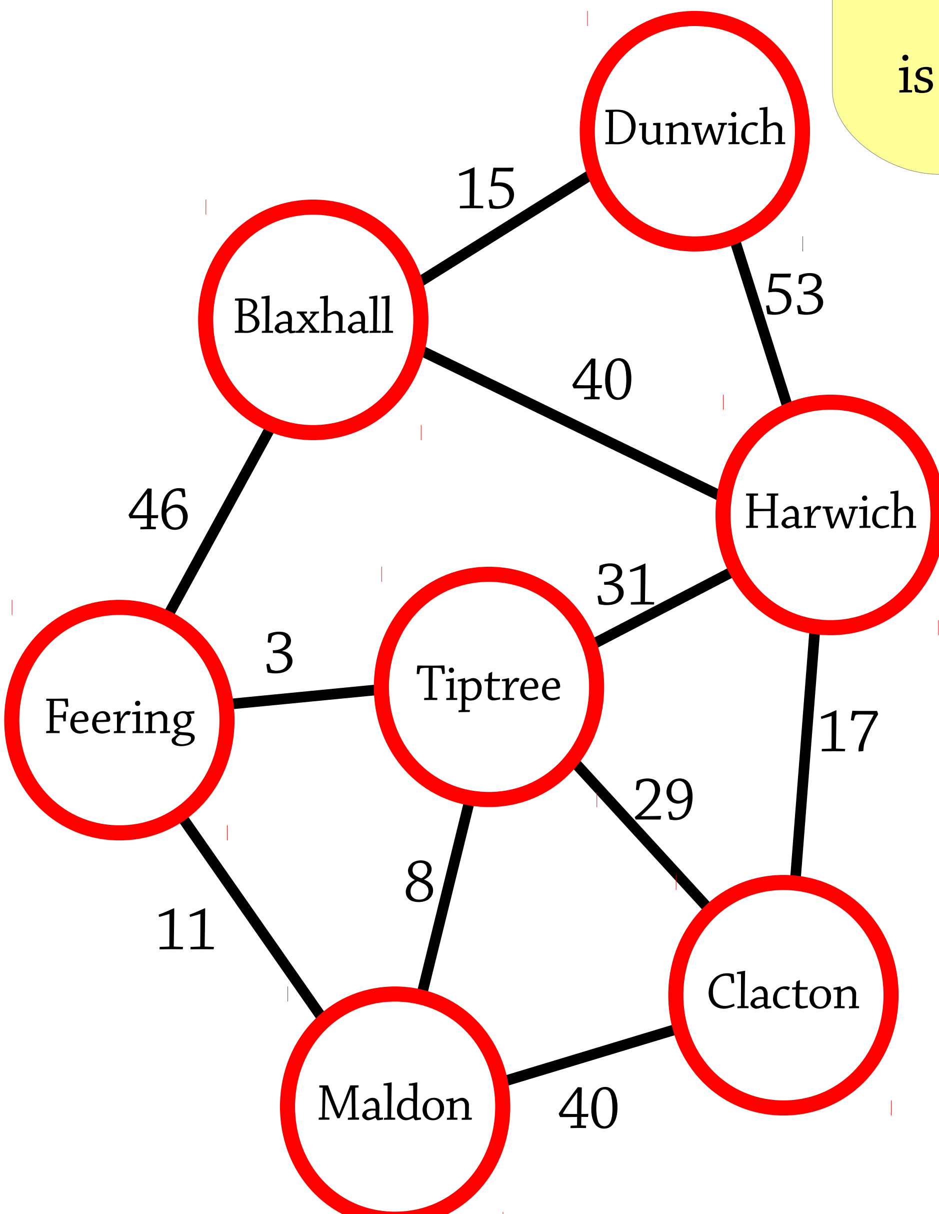
S = {Feering, Tiptree, Maldon, Clacton}
Lowest-weight edge from S to not-S is Clacton → Harwich

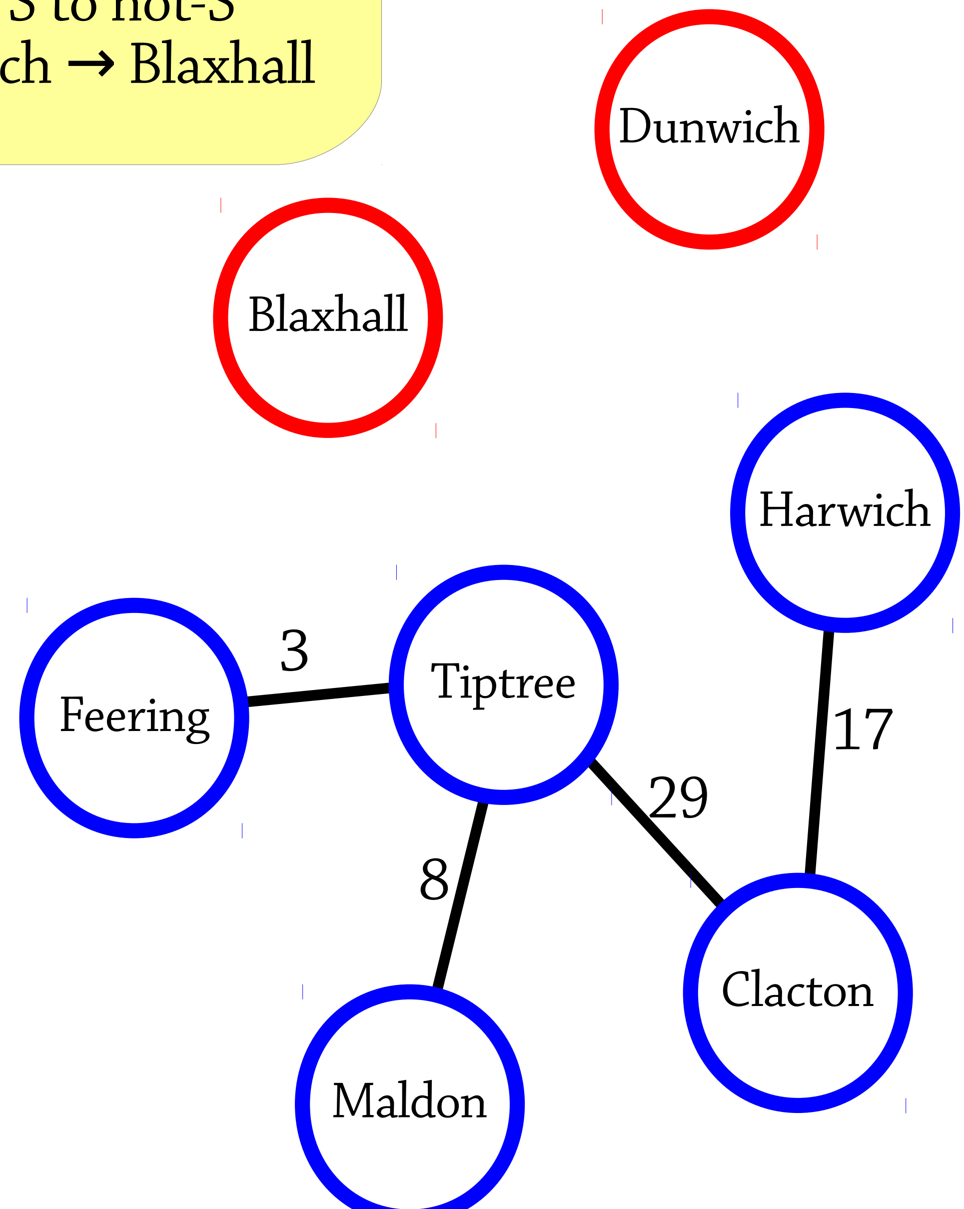




Minimum ees

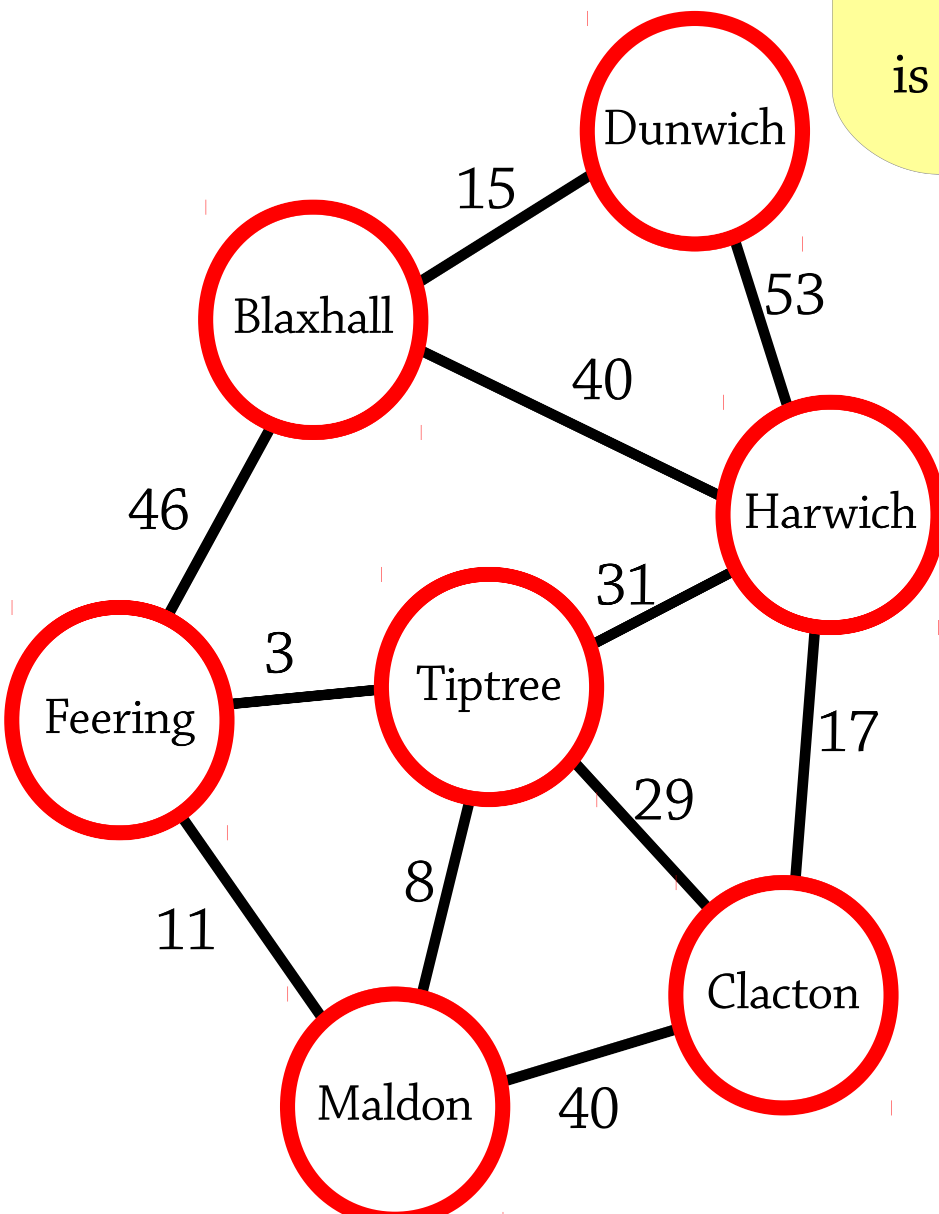
S = {Feering, Tiptree, Maldon, Clacton, Harwich}
Lowest-weight edge from S to not-S
is Harwich → Blaxhall

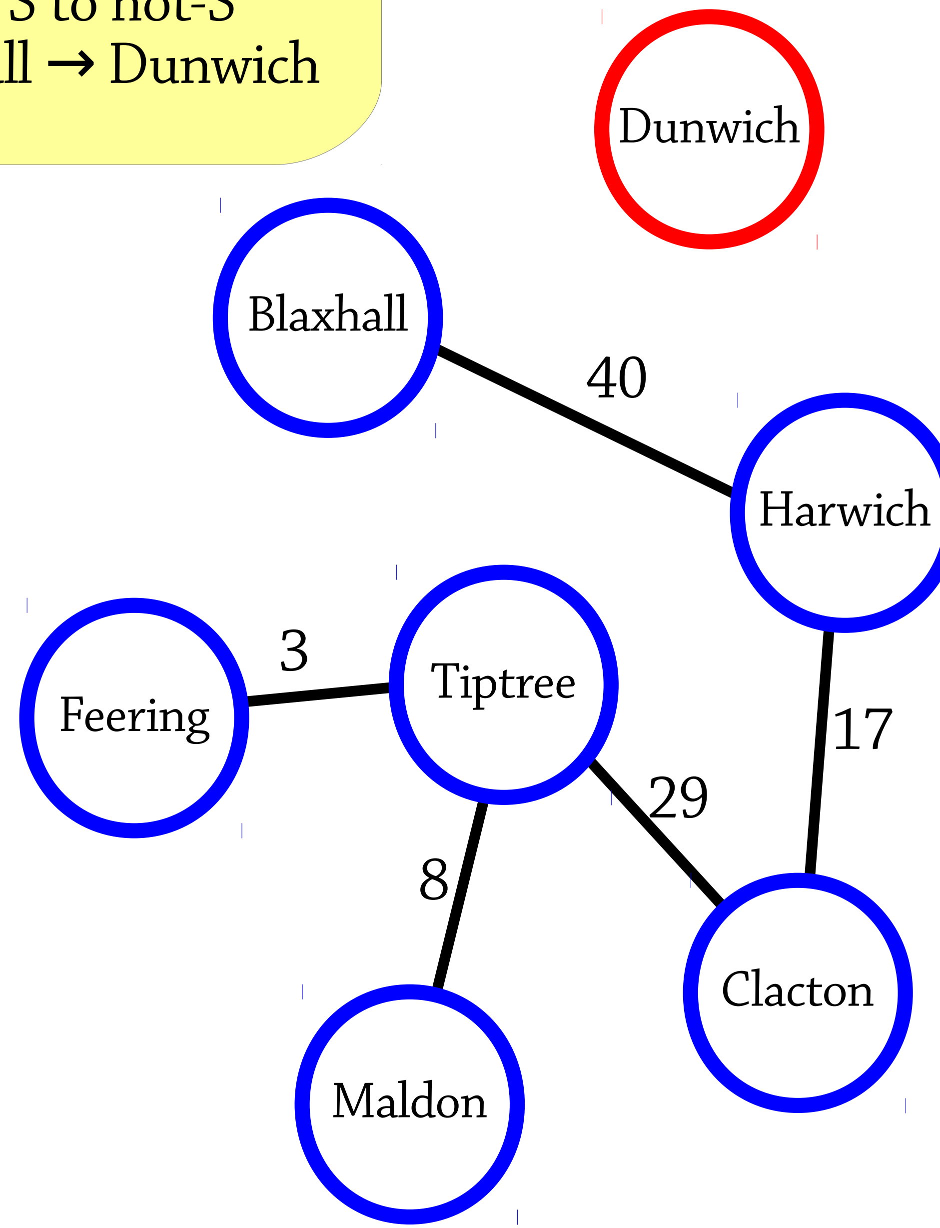




Minimun ees

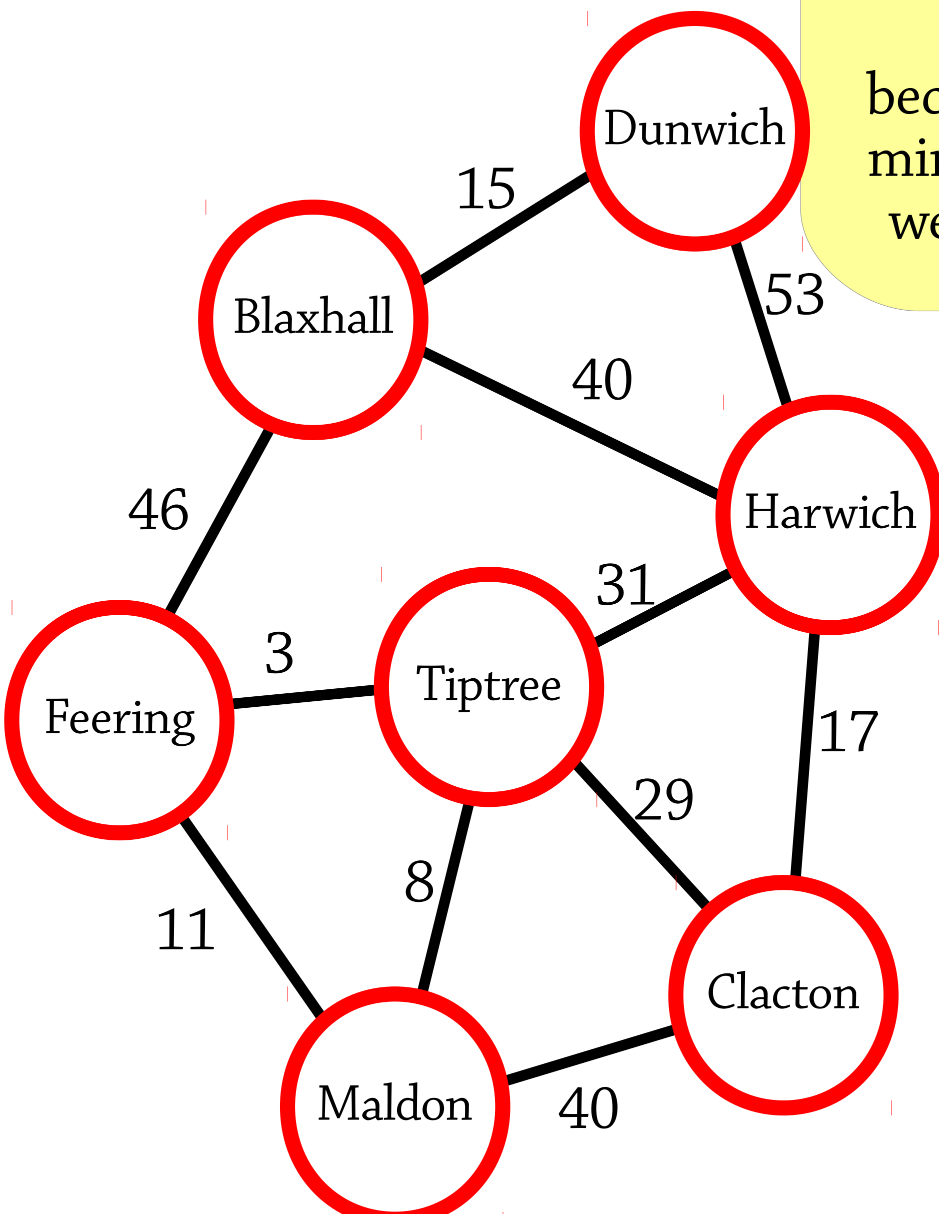
S = {Feering, Tiptree, Maldon, Clacton, Harwich, Blaxhall}
Lowest-weight edge from S to not-S is Blaxhall → Dunwich

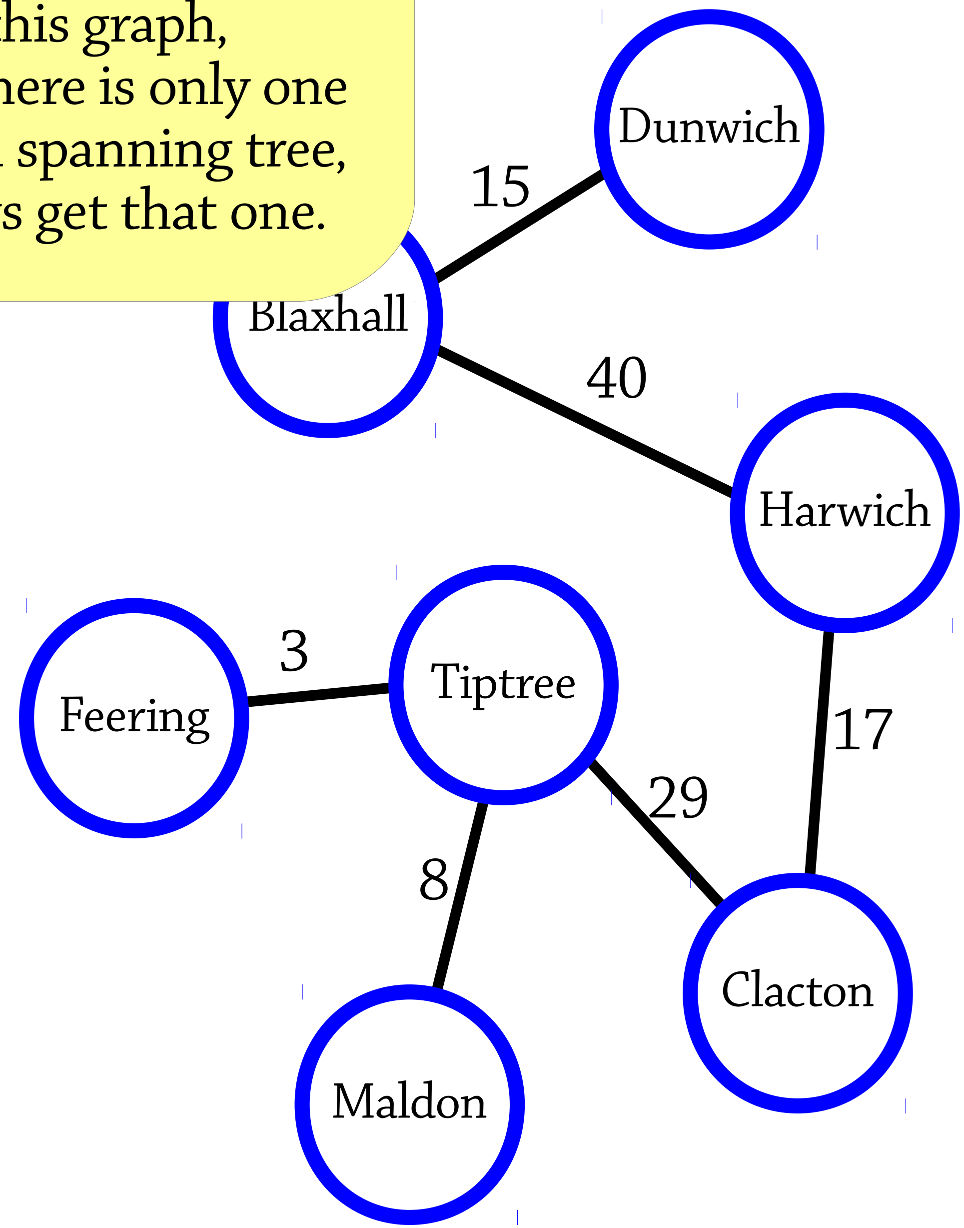




Minimu… ees

Notice:
we get a minimum
spanning tree
whatever node we start at!
For this graph,
because there is only one
minimum spanning tree,
we always get that one.





Prim's algorithm, efficiently

The operation

- Pick the *lowest-weight* edge between a node in S and a node not in S

takes O(n) time if we're not careful! Then Prim's algorithm will be $O(n^2)$

To implement Prim's algorithm, use a priority queue containing all edges between S and not-S

- Whenever you add a node to S, add all of its edges to nodes in not-S to a priority queue
- To find the lowest-weight edge, just find the minimum element of the priority queue
- Just like in Dijkstra's algorithm, the priority queue might return an edge between two elements that are now in S: ignore it

New time: O(n log n) :)

Summary

Breadth-first search – finding shortest paths in unweighted graphs, using a queue

Dijkstra's algorithm – finding shortest paths in weighted graphs – some extensions for those interested:

- Bellman-Ford: works when weights are negative
- A* – faster – tries to move *towards* the target node, where Dijkstra's algorithm explores equally in all directions

Prim's algorithm – finding minimum spanning trees

Both are *greedy algorithms* – they repeatedly find the “best” next element

- Common style of algorithm design

Both use a priority queue to get O(n log n)

- Dijkstra's algorithm is sort of BFS but using a priority queue instead of a queue

Many many many more graph algorithms

A* search
(not on exam)

A problem with Dijkstra's algorithm

We can use Dijkstra's algorithm to find the shortest route from A to B

But it explores *all* nodes in the graph that are closer than B!

A person planning a route would try to move *towards* B

Gothenburg to Stockholm?

The A* algorithm

Often we have a notion of *distance* in a graph

- e.g., Gothenburg to Stockholm is 400km as the crow flies
- No possible route can be shorter than this!

A* uses distance to guide the search

- Try to pick edges that reduce the distance to the target, avoid edges that increase the distance
- But still guaranteeing to find the shortest path!

The A* algorithm

We assume there is a function *h(x)* (the *heuristic*)

- In our example, h(x) is the distance from x to Stockholm as the crow flies

When we take an edge x → y, we are interested not only in the weight but also h(y)-h(x)

- If h(y)-h(x) is positive, we moved *away* from the target (bad); if it's negative, we moved *towards* the target (good)

To exploit h(y)-h(x), we take the input graph, and modify the weights of all the edges

- If we have an edge from *x* to *y*, we increase its weight by h(y)-h(x) – so "good" edges get cheaper and "bad" edges get more expensive

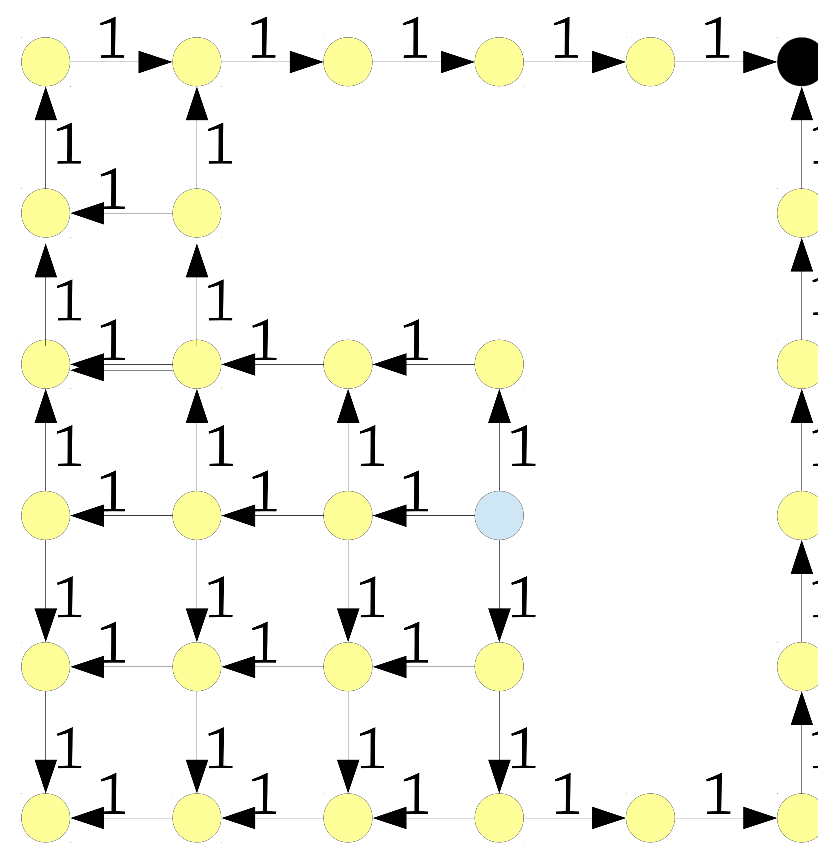
Then we run Dijkstra's algorithm on this new graph!

A* – an example

A* was originally invented for robot motion planning! Here is a floor with an obstacle in. (Edges given directions for simplicity.)

The robot wants to get from the blue node to the black node.

The shortest path has weight 9 – Dijkstra's algorithm will explore the whole graph!



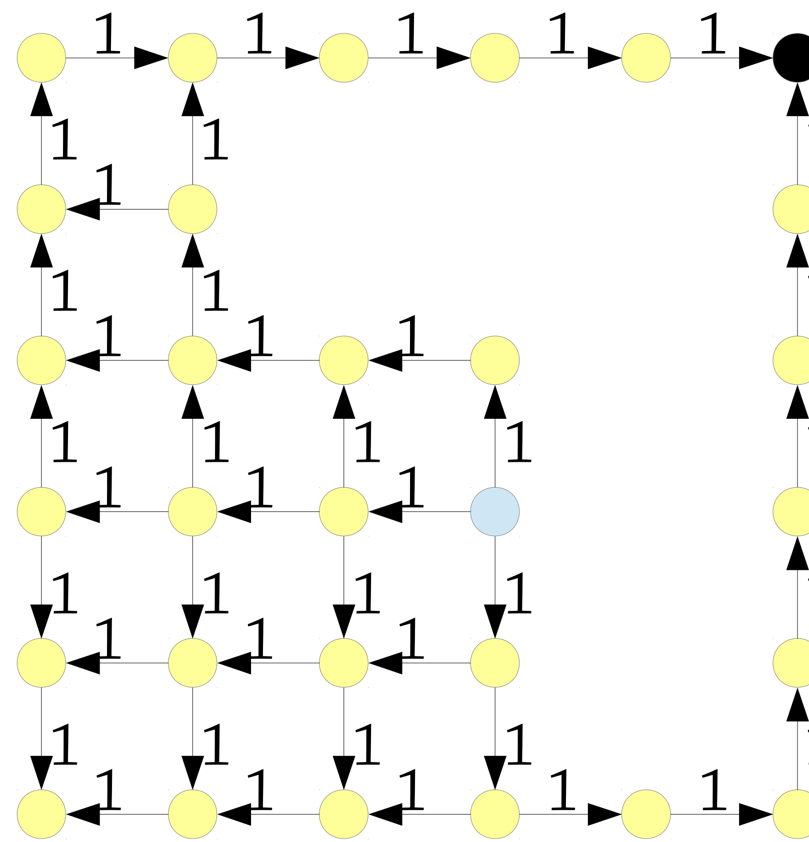
A* – an example

Now let's use the heuristic h(x) = x distance to black node + y distance to black node

e.g., h(blue node) = 5

If there is an edge from x to y, we add h(y)-h(x), so for this graph:

- If the edge goes up or right, we decrease its weight by 1
- If it goes down of left, we increase its weight by 1

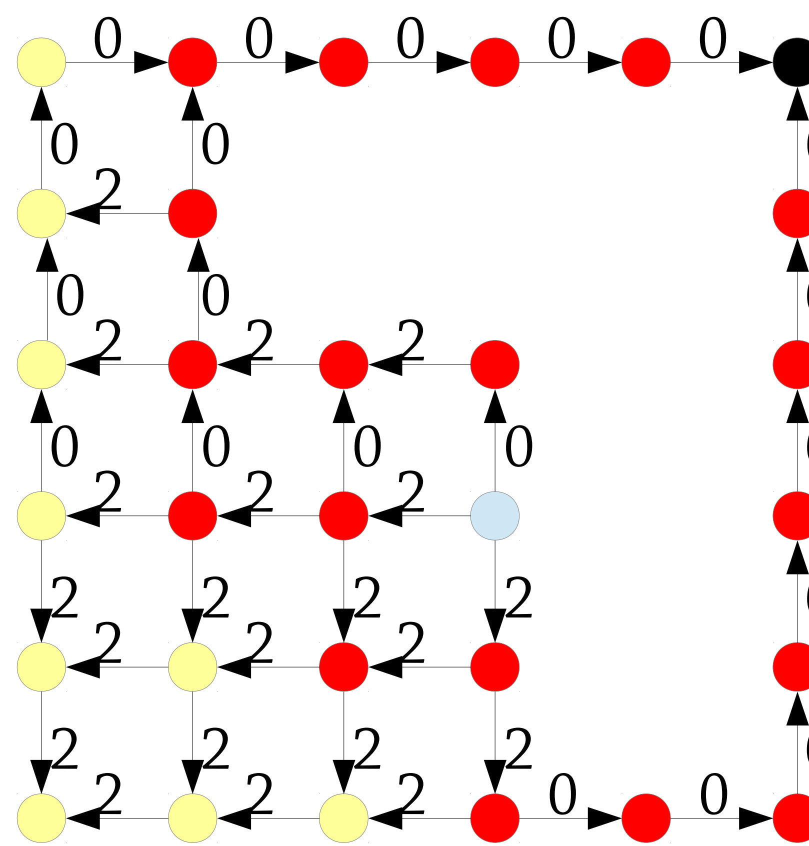


A* – an example

In the new graph, the up and right edges have weight 0, and the left and down edges have weight 2

The shortest path has weight 4 – you have to go left twice

The area the algorithm explores is highlighted in red



A* – why does it work?

In A*, we change the weights of all the edges – are we still going to get the shortest path for the original graph? Yes!

Suppose we have a path e.g. $a \rightarrow b \rightarrow c$, and weights w_{ab}, w_{bc} – the total weight of the path is $w_{ab} + w_{bc}$

Using A*, the weights are $w_{ab} + h(b) - h(a)$ and $w_{bc} + h(c) - h(b)$

The new weight of the path $a \rightarrow b \rightarrow c$ is:
$w_{ab} + h(b) - h(a) + w_{bc} + h(c) - h(b) = w_{ab} + w_{bc} + h(c) - h(a)$

So the total weight of each path from *source* to *target* is increased by h(target) - h(source) – a constant

The weight of each path changes, but by the same amount – so the shortest path is still the shortest path!

Some technicalities

Dijkstra's algorithm doesn't work if there is an edge with a negative weight

So we'd better be sure that modifying the weights never makes them negative

If we have an edge from x to y of weight w, the new weight is $w+h(y)-h(x)$, so this is fine as long as:

- $h(x) \leq w + h(y)$

That is, by following an edge you can't reduce the distance to the target by more than the weight of that edge – this is true e.g. of distance in maps

A* – summary

An extension of Dijkstra's algorithm that uses distance information to move *towards* the destination instead of exploring in all directions

- Still guaranteed to find the shortest path

Works very well in practice!

If we multiply the heuristic function by a constant, we can direct the search less or more aggressively

- But if we're too aggressive and the heuristic function returns too large values, the edge weights will become negative
- In this case we can't use Dijkstra's algorithm, but there is a more complex version of A^* we can use instead
- But this aggressive version of A^* can find suboptimal paths